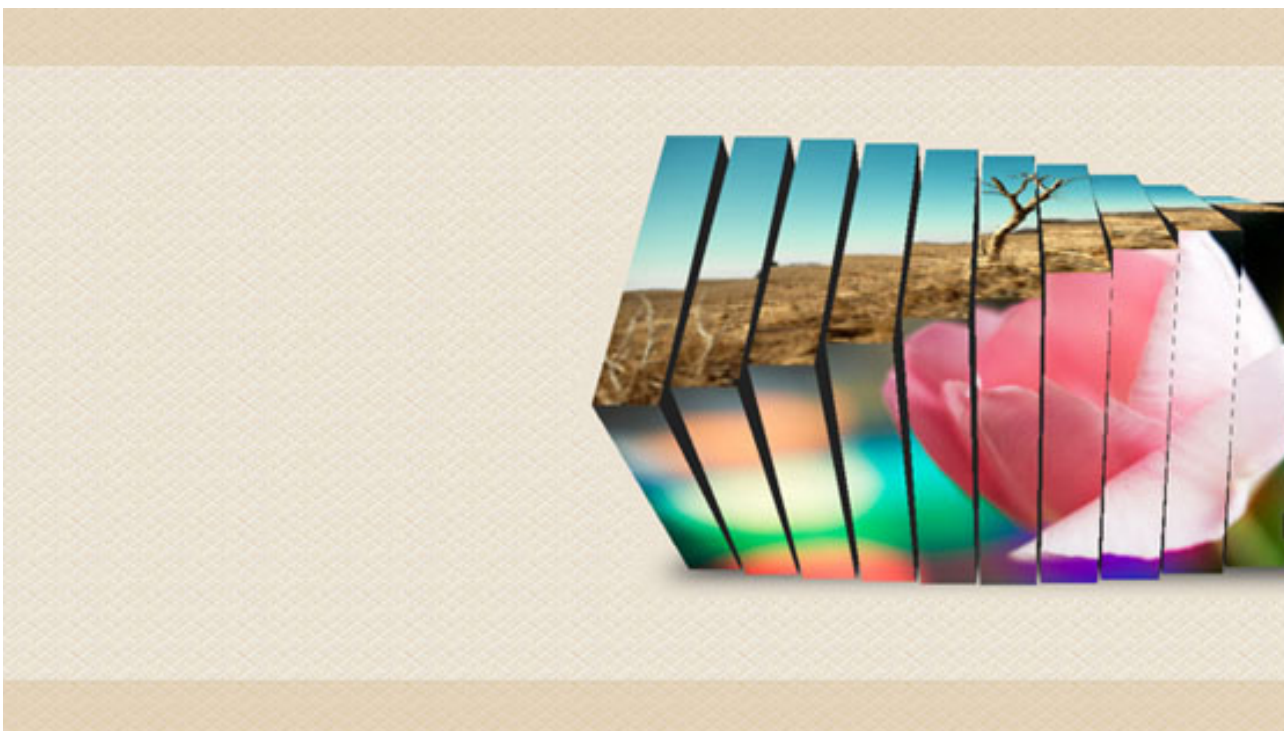
FONDEMENTS DU MULTIMÉDIA
TRANSFORMATIONS GÉOMÉTRIQUES



Kévin Rignault

Fondements du Multimédia - Compte Rendu
Master 1 Produits et Services Multimédia
Année Universitaire 2013-2014

Pays de Montbéliard Agglomération
UFC Université de Franche-Comté
STGI UFR Sciences Techniques Gestion de l'Industrie
PSM Produits et Services Multimédia

SOMMAIRE

INTRODUCTION

Au cours de ce premier semestre de Master Produits et Services Multimédia, nous utilisons particulièrement les transformations géométriques dans le cadre de la synthèse d'images notamment, que ce soit pour de la modélisation 3D ou du WebGL. Mais nous avons également pu les utiliser lors du développement d'un jeu de labyrinthe en Java.

Nous avons donc pu voir que les transformations géométriques sont très souvent utilisées afin de déplacer des éléments, de les transformer ou encore de créer une interactivité.

Plus généralement, ces transformations géométriques sont très utilisées dans le domaine de l'informatique. Derrière chaque logiciel, chaque jeu se cachent des transformations géométriques plus ou moins complexes, et même derrière le pointeur de la souris qui ne se déplacerait pas sans ces transformations.

C'est pourquoi, au cours de ce premier semestre, nous nous sommes penchés sur les transformations géométriques et leurs implémentations en ActionScript3 puis Processing, afin de comprendre leur fonctionnnement en détails. Nous avons notamment créé des programmes permettant de comprendre les fonctions trigonométriques ainsi que les transformations géométriques 2D et 3D.

PRÉLIMINAIRES MATHÉMATIQUES

Avant d'expliquer ce que sont les transformations géométriques, il est nécessaire de connaitre des notions mathématiques fondamentales à leur mise en place.

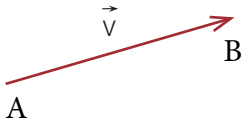
POINTS ET VECTEURS

GÉNÉRALITÉ

Le point est sans doute l'élément mathématique le plus utilisé en géométrie et pourtant, il est invisible. Généralement on le représente par un petit point, mais sa seule caractéristique réelle est sa position.

Si l'on considère le couple de points (A,B), on peut y associer un vecteur $\vec{v}$. Ainsi, on parle du vecteur $\overrightarrow{AB}$. Le vecteur est un élément mathématique caractérisé par :

- **sa direction**. *Ici, celle de la droite (AB)*
- **son sens**. *Ici, de A vers B*
- **sa norme**. *Ici, c'est la distance AB, notée* $\|\vec{v}\|$



Dans un espace 2D, un vecteur est noté $\vec{v}\begin{pmatrix} x \\ y \end{pmatrix}$ où x et y sont les coordonnées du vecteur. On parle aussi de composantes du vecteur. En 3D, une troisième composante *z* s'ajoute.

* On dit d'un vecteur qu'il est normé lorsque sa norme est égale à 1. On note $\|\vec{v}\|=1$.

Quelques exemples de vecteurs normés pratiques

Dans un espace 2D

$$\vec{v}\begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad \vec{v}\begin{pmatrix} 0 \\ 1 \end{pmatrix} \quad \vec{v}\begin{pmatrix} -1 \\ 0 \end{pmatrix} \quad \vec{v}\begin{pmatrix} 0 \\ -1 \end{pmatrix}$$

Dans un espace 3D

$$\vec{v}\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \quad \vec{v}\begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \quad \vec{v}\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \quad \vec{v}\begin{pmatrix} -1 \\ 0 \\ 0 \end{pmatrix} \quad \vec{v}\begin{pmatrix} 0 \\ -1 \\ 0 \end{pmatrix} \quad \vec{v}\begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix}$$

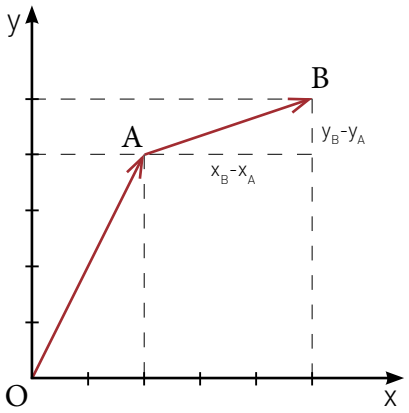
PRINCIPALES OPÉRATIONS

- **Distance entre deux points et norme d'un vecteur**

On considère un repère orthonormal (O; x, y) et deux points $A(x_A,y_A)$ et $B(x_B,y_B)$.

La distance entre A et B est[1] :

$$AB = \| \overrightarrow{AB} \| = \sqrt{(x_B - x_A)^2 + (y_B - y_A)^2}$$



* La distance entre un point A(x,y) et l'origine (0,0) est :

$$OA = \| \overrightarrow{OA} \| = \sqrt{(x_A - 0)^2 + (y_A - 0)^2}$$
$$= \sqrt{x^2 + y^2}$$

- **Additions de vecteurs**

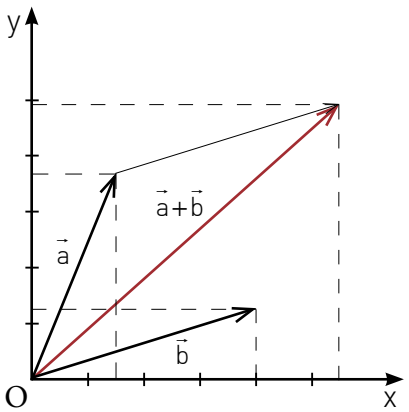
On considère deux vecteurs $\vec{a}\begin{pmatrix} x_a \\ y_a \end{pmatrix}$ et $\vec{b}\begin{pmatrix} x_b \\ y_b \end{pmatrix}$.

L'addition de ces deux vecteurs est :

$$\vec{a}+\vec{b}\begin{pmatrix} x_a + x_b \\ y_a + y_b \end{pmatrix}$$

Il est également possible de représenter le résultat d'une addition de vecteurs en utilisant la règle du parallélogramme.

Schéma ci-contre
** Dans cet exemple, on place le vecteur b à l'extrémité du vecteur a.*



[1] Théorème de Pythagore

- Multiplication d'un vecteur par un réel

On considère un réel λ non nul et un vecteur $\vec{v}\begin{pmatrix} x \\ y \end{pmatrix}$ non nul[1] également.

La représentation d'une multiplication d'un vecteur par un réel peut s'exprimer telle que :

f: R x E -> E. Où R est l'espace réel et E l'espace vectoriel[2]

Le résultat de $\lambda\vec{v}$ est un donc vecteur tel que :

$$\lambda\vec{v}\begin{pmatrix} \lambda x \\ \lambda y \end{pmatrix}$$

Dans ce cas, le vecteur $\lambda\vec{v}$ a :
- la même direction que $\vec{v}$
- le même sens que $\vec{v}$ si λ est strictement positif
- un sens opposé à celui de $\vec{v}$ si λ est strictement négatif
- une norme telle que $\|\lambda\vec{v}\| = |\lambda| . \|\vec{v}\|$

Quelques règles de calcul

En considérant deux réels α et β et deux vecteurs $\vec{u}$ et $\vec{v}$, on peut établir les règles de calcul suivantes :

- Distributivité
$\alpha(\vec{u} + \vec{v}) = \alpha\vec{u} + \alpha\vec{v}$
$(\alpha + \beta)\vec{u} = \alpha\vec{u} + \beta\vec{u}$

- Associativité
$\alpha(\beta\vec{u}) = (\alpha\beta)\vec{u}$
$0\vec{u} = \vec{0}$

Combinaison linéaire

La combinaison linéaire consiste à reprendre les opérations précédentes, à savoir l'addition et la multiplication. Elle peut être représentée de la manière suivante :

$$\alpha_1\vec{v}_1 + \alpha_2\vec{v}_2 + \ldots + \alpha_n\vec{v}_n = \sum \alpha_i\vec{v}_i$$

[1] En considérant $\vec{v} = \overrightarrow{AB}$, $\vec{v}$ est un vecteur nul si et seulement si A = B. On note $\vec{v} = \vec{0}$.

[2] On parle de loi de composition externe

- Produit scalaire

On considère deux vecteurs $\vec{u}\begin{pmatrix} x \\ y \end{pmatrix}$ et $\vec{v}\begin{pmatrix} x' \\ y' \end{pmatrix}$.

La représentation d'une multiplication de deux vecteurs peut s'exprimer telle que :

f: E x E = R. Où E est l'espace vectoriel et R l'espace réel

Le résultat de $\vec{u} \cdot \vec{v}$ est donc un réel tel que :

$$\vec{u} \cdot \vec{v} = xx' + yy'$$

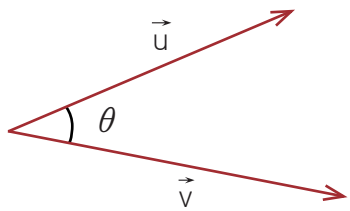
- Produit scalaire et angles

En considérant deux vecteurs $\vec{u}$ et $\vec{v}$, on peut définir la relation suivante :

$$\vec{u} \cdot \vec{v} = ||\vec{u}|| \times ||\vec{v}|| \times \cos(\theta)$$

ou encore

$$\cos(\theta) = \frac{\vec{u}.\vec{v}}{||\vec{u}||.||\vec{v}||}$$



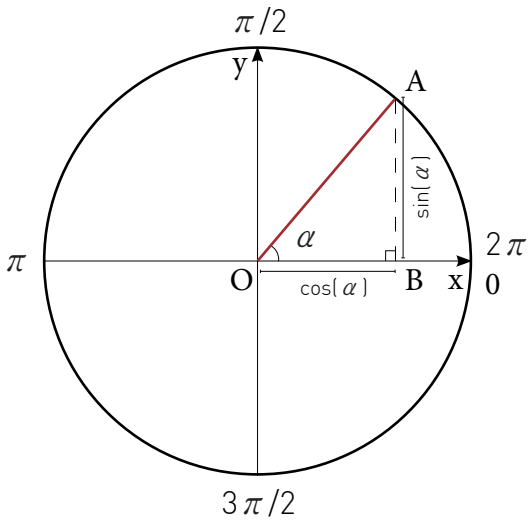
Par conséquent, si $\vec{u} \cdot \vec{v} = 0$ alors $\cos(\theta) = 0$.
Or, si $\cos(\theta) = 0$ alors $\theta = 90$.
Donc $\vec{u}$ et $\vec{v}$ sont orthogonaux.

On peut ainsi dire que deux vecteurs sont **orthogonaux**, si et seulement si, leur **produit scalaire est nul.**

FONCTIONS TRIGONOMÉTRIQUES

Les fonctions trigonométriques permettent de calculer des angles et/ou des longueurs notamment. Pour cela, on utilise un cercle trigonométrique -dont les valeurs pour un tour de cercle[1] sont comprises entre 0 et 2π - inscrit sur un repère orthonormé.

On considère trois points O, A et B placés sur le repère orthonormé et le cercle trigonométrique ci-dessous. En projetant le point A sur l'axe Ox, on obtient un triangle rectangle en B, ce qui nous permet d'établir les relations suivantes :



$$\sin(\alpha) = \frac{\text{côté opposé}}{\text{hypoténuse}}$$

$$\cos(\alpha) = \frac{\text{côté adjacent}}{\text{hypoténuse}}$$

$$\tan(\alpha) = \frac{\text{côté opposé}}{\text{côté adjacent}}$$

avec OA l'hypoténuse
OB le côté adjacent
AB le côté opposé

Valeurs de sinus pratiques

$\sin 0 = 0$
$\sin \pi/2 = 1$
$\sin \pi = 0$
$\sin -\pi/2 = -1$

D'où $\sin \alpha \in [-1,1]$

Et $\sin(-\alpha) = -\sin\alpha$
-> Fonction impaire

Valeurs de cosinus pratiques

$\cos 0 = 1$
$\cos \pi/2 = 0$
$\cos \pi = -1$
$\cos -\pi/2 = 0$

D'où $\cos \alpha \in [-1,1]$

Et $\cos(-\alpha) = \cos\alpha$
-> Fonction paire

[1] Dans le sens inverse des aiguilles d'une montre

Règles de calcul

sin(a+b) = sin(a)cos(b) + cos(a)sin(b)
cos(a+b) = cos(a)cos(b) - sin(a)sin(b)

À l'aide de ces règles de calcul, nous allons pouvoir établir une relation permettant de faire une rotation d'un point sur un cercle.

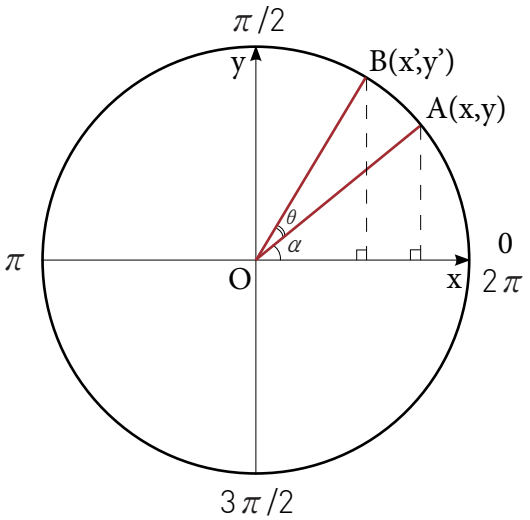
En considérant un point A de coordonnées (x,y) et un point B de coordonnées (x',y'), on peut exprimer les relation suivantes :

On sait que
$x = \cos \alpha$ et $y = \sin \alpha$

D'où
$x' = \cos(\alpha+\theta)$ et $y' = \sin(\alpha+\theta)$

Soit
$x' = \cos(\alpha)\cos(\theta) - \sin(\alpha)\sin(\theta)$
$x' = x.\cos(\theta) - y.\sin(\theta)$

et
$y' = \sin(\alpha)\cos(\theta) + \cos(\alpha)\sin(\theta)$
$y' = y.\cos(\theta) + x.\sin(\theta)$



En factorisant, on peut donc exprimer cette rotation sous la forme suivante :

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$ [1]

Il s'agit d'une représentation sous forme de matrices, que nous allons voir juste à la suite.

[1] Nous venons d'établir la matrice de rotation

MATRICES

GÉNÉRALITÉ

Une matrice se présente comme un tableau de nombres à n lignes et m colonnes. La dimension de la matrice est donc définie par le couple (n,m). Chaque élément de la matrice est désigné par deux indices[1], à savoir son numéro de ligne suivi de son numéro de colonne. On le note donc a_{ij}.

Ainsi, on peut représenter une matrice comme suit :

$$\begin{bmatrix} a_{11} & \dots & a_{1m} \\ \dots & \dots & \dots \\ a_{n1} & \dots & a_{nm} \end{bmatrix}$$

Exemple

$$\begin{bmatrix} 1 & 5 & 9 \\ 3 & 7 & 8 \end{bmatrix}$$

Ici, il s'agit d'une matrice de dimension (2,3). Par exemple, l'élément en (2,1) est 3.

Les matrices peuvent être de toutes les tailles, par conséquent, certaines matrices se distinguent de par leurs dimensions.

Cas particuliers
Une matrice à 1 ligne et m colonnes est une **matrice ligne** (ou vecteur ligne).

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \end{bmatrix}$$

Une matrice à n lignes et 1 colonne est une **matrice colonne** (ou vecteur colonne).

$$\begin{bmatrix} a_{11} \\ a_{21} \\ a_{31} \end{bmatrix}$$

[1] par convention, en mathématique les indices débutent à 1, en informatique à 0

PRINCIPALES OPÉRATIONS

Comme pour les vecteurs, il est possible d'effectuer certaines opérations sur les matrices.

- Addition / Soustraction de matrices
L'addition est possible si et seulement si les matrices sont de mêmes dimensions. La somme de deux matrices A et B est une nouvelle matrice C de même dimension obtenue en additionnant les éléments des deux matrices correspondants.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{bmatrix} = \begin{bmatrix} a_{11}+b_{11} & a_{12}+b_{12} & a_{13}+b_{13} \\ a_{21}+b_{21} & a_{22}+b_{22} & a_{23}+b_{23} \end{bmatrix}$$

Exemple

$$\begin{bmatrix} 3 & 1 & 5 \\ 7 & 4 & 2 \end{bmatrix} + \begin{bmatrix} 6 & 2 & 8 \\ 9 & 0 & 4 \end{bmatrix} = \begin{bmatrix} 9 & 3 & 13 \\ 16 & 4 & 6 \end{bmatrix}$$

La soustraction se déroule de la même manière. La différence de deux matrices A et B est donc une nouvelle matrice C de même dimension obtenue en soustrayant les éléments des deux matrices correspondants.

- Multiplication d'une matrice par un nombre
La multiplication d'une matrice par un nombre consiste à multiplier tous les éléments de la matrice par le nombre. Le produit obtenu est donc une nouvelle matrice de même dimension.

$$k \times \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} = \begin{bmatrix} k.a_{11} & k.a_{12} & k.a_{13} \\ k.a_{21} & k.a_{22} & k.a_{23} \end{bmatrix}$$

Exemple

$$2 \times \begin{bmatrix} 5 & 3 & 1 \\ 2 & 4 & 6 \end{bmatrix} = \begin{bmatrix} 10 & 6 & 2 \\ 4 & 8 & 12 \end{bmatrix}$$

- Multiplication de matrices

La multiplication de deux matrices est possible si et seulement le nombre de colonnes de la première matrice est égale au nombre de lignes de la deuxième. Le produit de deux matrices A et B est donc une nouvelle matrice C dont le nombre de lignes est égale à celui de la première matrice, et le nombre de colonnes égale à celui de la deuxième tel que :

$$A_{n,m} \times B_{m,p} = C_{n,p}$$

Ainsi, la multiplication de matrices n'est pas commutatives : A x B ≠ B x A.
Cette nouvelle matrice est obtenue en multipliant les éléments d'une ligne de la première matrice par les éléments de la colonne correspondante de la seconde.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{bmatrix} = \begin{bmatrix} a_{11} \times b_{11} + a_{12} \times b_{21} + a_{13} \times b_{31} & a_{11} \times b_{12} + a_{12} \times b_{22} + a_{13} \times b_{32} \\ a_{21} \times b_{11} + a_{22} \times b_{21} + a_{23} \times b_{31} & a_{21} \times b_{12} + a_{22} \times b_{22} + a_{23} \times b_{32} \end{bmatrix}$$

Exemple

$$\begin{bmatrix} 2 & 1 & 4 \\ 3 & 5 & 2 \end{bmatrix} \times \begin{bmatrix} 1 & 2 \\ 3 & 1 \\ 2 & 4 \end{bmatrix} = \begin{bmatrix} 2 \times 1 + 1 \times 3 + 4 \times 2 & 2 \times 2 + 1 \times 1 + 4 \times 4 \\ 3 \times 1 + 5 \times 3 + 2 \times 2 & 3 \times 2 + 5 \times 1 + 2 \times 4 \end{bmatrix}$$

Donc

$$\underset{(2,3)}{\begin{bmatrix} 2 & 1 & 4 \\ 3 & 5 & 2 \end{bmatrix}} \times \underset{(3,2)}{\begin{bmatrix} 1 & 2 \\ 3 & 1 \\ 2 & 4 \end{bmatrix}} = \underset{(2,2)}{\begin{bmatrix} 13 & 21 \\ 22 & 19 \end{bmatrix}}$$

Maintenant que nous avons expliqué ce qu'étaient les matrices, nous allons voir quel peut être leur intérêt au sein des transformations géométriques. Mais avant ça, nous allons présenter quelques unes de ces transformations dans l'espace 2D.

TRANSFORMATIONS GÉOMÉTRIQUES

Les transformations géométriques permettent principalement de déplacer un objet dans une scène, répliquer un objet ou bien encore le déformer. Nous allons aborder la translation, la rotation, le changement d'échelle et le cisaillement.

TRANSFORMATIONS 2D

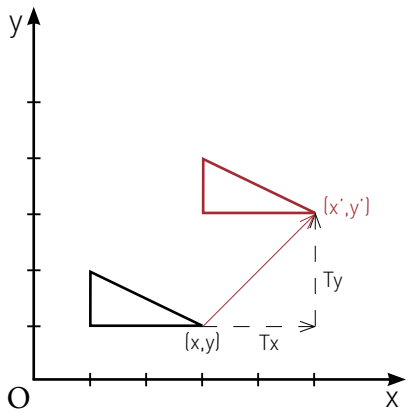
TRANSLATION

La translation d'un objet consiste à déplacer tous les points de cet objet sur l'axe x et/ou y selon une même distance, une même direction et un même sens, c'est à dire selon un vecteur. La translation conserve ainsi les distances et les angles, par conséquent l'objet obtenu est identique à l'objet initial, seuls ses points de coordonnées diffèrent.

On peut dont exprimer les relations suivantes :

$$x' = x + Tx$$
$$y' = y + Ty$$

où Tx et Ty représente respectivement les déplacement en x et en y de la translation.



* La translation correpond donc à une addition de vecteurs représentée comme suit :

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix}$$

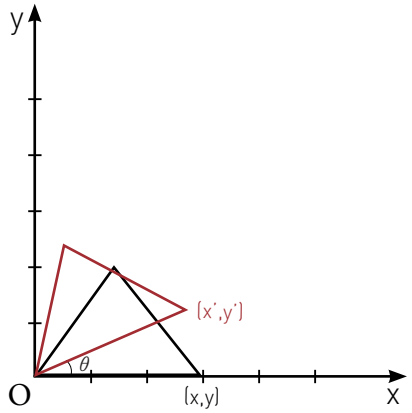
ROTATION

La rotation d'un objet consiste à faire pivoter tous les points de cet objet autour d'un point et selon un angle. Comme la translation, la rotation conserve les distances et les angles. Ici encore, l'objet obtenu est identique à l'objet initial, seuls ses points de coordonnées diffèrent. Dans la partie *«Fonctions Trigonométriques»*, nous avons démontré comment faire pivoter un point sur un cercle. Nous allons nous appuyer sur cette démonstration pour exprimer les relations permettant d'effectuer une rotation.

Nous avons vu que :

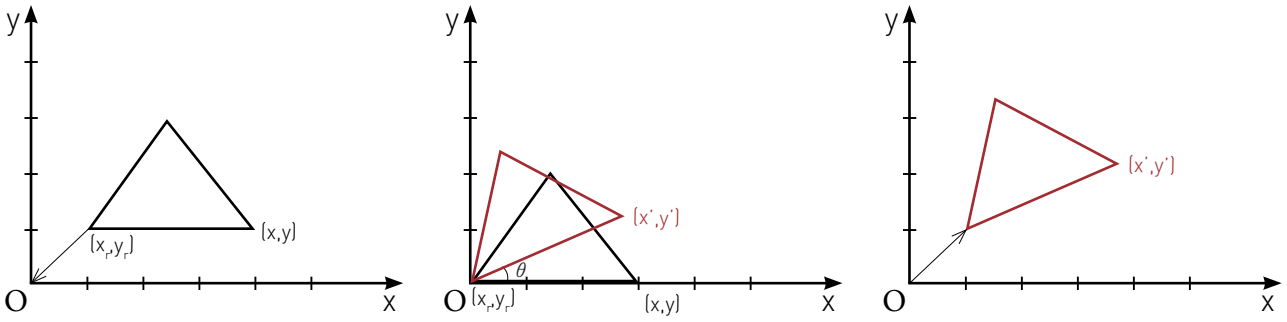
$$x' = x.\cos(\theta) - y.\sin(\theta)$$
$$y' = y.\cos(\theta) + x.\sin(\theta)$$

où θ est l'angle de rotation dans le sens anti-horaire.



* Cette expression est en fait un cas particulier qui permet d'effectuer une rotation de l'objet **autour de l'origine**.

Le cas général permet d'effectuer une rotation **autour d'un point fixe**. Pour faire cela, on applique une translation à l'origine, on applique ensuite la rotation, et pour finir on applique la translation inverse pour replacer l'objet.



On peut ainsi établir les relation suivantes :

$$x' = x_r + (x - x_r)\cos(\theta) - (y - y_r)\sin(\theta)$$
$$y' = y_r + (y - y_r)\cos(\theta) + (x - x_r)\sin(\theta)$$

où θ est l'angle de rotation dans le sens anti-horaire.

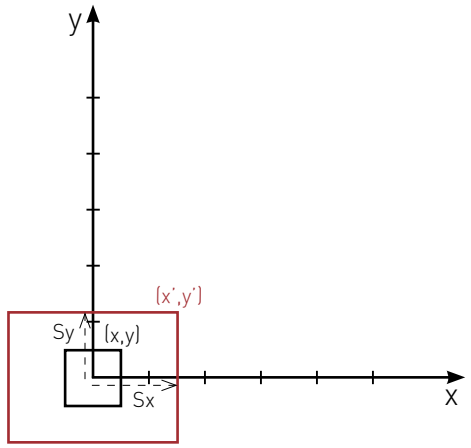
CHANGEMENT D'ECHELLE

Le changement d'échelle d'un objet consiste à agrandir et/ou réduire l'objet en x et/ou en y par rapport à un point. Dans le cas où le changement d'échelle est le même sur x et sur y, on parle alors d'homothétie.

Ainsi, on peut exprimer les relations suivantes :

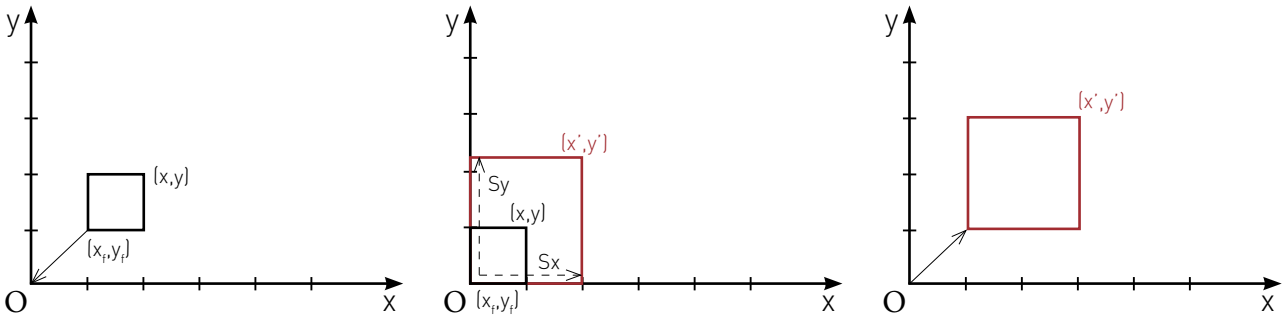
x' = x.Sx
y' = y.Sy

où Sx et Sy sont les facteurs :
- d'agrandissement si >1
- de réduction si <1



* Comme pour la rotation, cette expression est en fait un cas particulier qui permet d'effectuer une rotation de l'objet **autour de l'origine**.

Par conséquent, comme pour la rotation, le cas général permet d'effectuer un changement d'échelle **par rapport à un point fixe**. Pour faire cela, on applique une translation à l'origine, on applique ensuite le changement d'échelle, et pour finir on applique la translation inverse pour replacer l'objet.



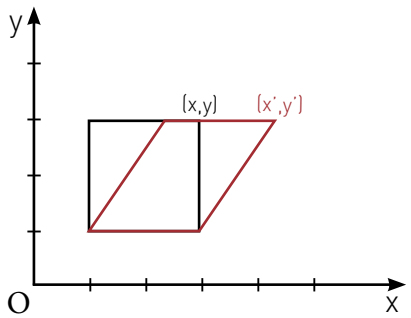
On peut ainsi établir les relation suivantes :

$x' = x_f + (x - x_f).Sx$
$y' = y_f + (y - y_f).Sy$

où θ est l'angle de rotation dans le sens anti-horaire.

CISAILLEMENT

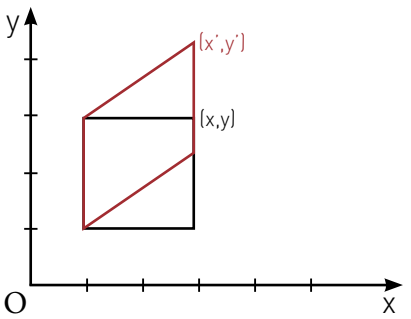
Le cisaillement d'un objet consiste à le déformer sur l'axe x et/ou y. On peut représenter trois types de cisaillement comme suit :



Le cissaillement horizontal
Déformation de l'objet selon les abcisses

On peut exprimer les relations suivantes :

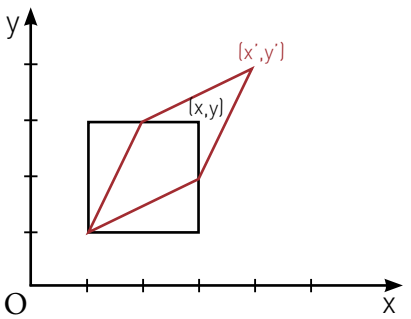
x' = x + y.Cx
y' = y



Le cissaillement vertical
Déformation de l'objet selon les ordonnées

On peut exprimer les relations suivantes :

x' = x
y' = y + x.Cy



Le cissaillement général
Déformation de l'objet selon les ordonnées

On peut exprimer les relations suivantes :

x' = x + Cx.y
y' = y + Cy.x

Maintenant que nous avons expliqué ce qu'étaient les transformations géométriques une à une, nous allons voir que chaque transformation possède sa propre matrice. Nous verrons par la suite que l'intérêt de représenter les transformations sous forme de matrices est de pouvoir combiner les transformations entre elles.

REPRÉSENTATION MATRICIELLE

MATRICE DE TRANSFORMATIONS

À l'aide des relations déjà évoquées ainsi que des règles de calcul sur les matrices, nous allons pouvoir établir les matrices de chaque transformation.

- Rotation
On sait que **x' = x.cos(θ) - y. sin(θ)** et **y' = y.cos(θ) + x.sin(θ)**
On peut donc établir une multiplication de matrices qui vérifie ce résultat :

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix}$$

- Changement d'échelle
On sait que **x' = x.Sx** et **y' = y.Sy**
On peut donc établir une multiplication de matrices qui vérifie ce résultat :

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} Sx & 0 \\ 0 & Sy \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix}$$

avec Sx=1, si le changement d'échelle s'effectue uniquement sur l'axe y
avec Sy=1, si le changement d'échelle s'effectue uniquement sur l'axe x

- Cisaillement
On sait que **x' = x + Cx.y** et **y' = y + Cy.x**
On peut donc établir une multiplication de matrices qui vérifie ce résultat :

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & Cx \\ Cy & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix}$$

avec Cx=0, si la déformation s'effectue uniquement sur l'axe y
avec Cy=0, si la déformation s'effectue uniquement sur l'axe x

Toutes ces transformations sont des transformations linéaires, c'est à dire qu'elles s'expriment sous la forme d'une fonction affine telle que **x' = a.x + b**. Cependant, la translation n'est pas une transformation linéaire puisqu'elle s'exprime sous la forme **x' = x + b**. En effet, pour la translation, on sait que **x' = x + Tx** et **y' = y + Ty**. Par conséquent, il n'existe pas, pour la translation, de représentation matricielle *compatible* avec les représentations matricielles des transformations précédentes.

Or, comme on l'expliquait plus haut, l'intérêt des matrices est de pouvoir les combiner entres elles. Nous allons donc devoir homogénéiser toutes les matrices, et pour ce faire nous allons passer par des coordonnées homogènes.

COORDONNÉES HOMOGÈNES ET MATRICES HOMOGÈNES

Afin d'obtenir des coordonnées homogènes, on ajoute une troisième coordonnée, w. Ainsi, un point 2D devient un vecteur à trois coordonnées. Dans notre cas, cette troisième composante sera toujours égale à 1.

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

À partir de cette représentation, nous allons pouvoir obtenir des matrices homogènes afin de les combiner entre elles, même la translation. En effet, on peut maintenant établir une multiplication de matrices permettant d'obtenir les expressions de la translation.

- Translation

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & Tx \\ 0 & 1 & Ty \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

On obtient ainsi :

x' = x.1 + y.0 + 1.Tx
x' = x + Tx

y' = x.0 + y.1 + 1.Ty
y' = y + Ty

On retrouve donc les expressions relatives à la translation. Il ne reste plus qu'à faire de mêmes pour les autres matrices.

- Rotation

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} x \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

On obtient ainsi :

$x' = x.\cos(\theta) - y.\sin(\theta) + 1.0$ $\quad$ $y' = x.\sin(\theta) + y.\cos(\theta) + 1.0$

$x' = x.\cos(\theta) - y.\sin(\theta)$ $\quad$ **$y' = x.\sin(\theta) + y.\cos(\theta)$**

-> On retrouve donc les expressions relatives à la rotation.

- Changement d'échelle

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} Sx & 0 & 0 \\ 0 & Sy & 0 \\ 0 & 0 & 1 \end{bmatrix} x \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

On obtient ainsi :

$x' = x.Sx + y.0 + 1.0$ $\quad$ $y' = x.0 + y.Sy + 1.0$

$x' = x.Sx$ $\quad$ **$y' = y.Sy$**

-> On retrouve donc les expressions relatives au changement d'échelle.

- Cisaillement

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & Cx & 0 \\ Cy & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} x \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

On obtient ainsi :

$x' = x.1 + y.Cx + 1.0$ $\quad$ $y' = x.Cy + y.1 + 1.0$

$x' = x + Cx.y$ $\quad$ **$y' = y + Cy.x$**

-> On retrouve donc les expressions relatives au cisaillement.

Maintenant que nous avons des matrices de transformations homogènes, de dimensions 3 par 3, nous allons pouvoir les combiner et ainsi pouvoir effectuer plusieurs transformations à la fois. C'est très utile en informatique dans le but d'optimiser le nombre de calcul notamment.

COMPOSITION DE TRANSFORMATIONS

La composition de transformations consiste à multiplier les matrices de transformations entre elles afin d'obtenir une nouvelle matrice qui contient l'ensemble de ces transformations.Il est possible de multiplier toutes les matrices de transformations entre elles puisqu'elles sont à présent toutes de même dimensions. Cependant, nous avons vu plus tôt que la multiplication de matrices n'est pas commutative. Par conséquent, l'ordre des multiplications -et donc des transformations- est important.

Exemple
Posons le produit matriciel permettant d'effectuer un changement d'échelle suivi d'une translation. On note :

$$T(Tx,Ty) \times S(Sx, Sy) = \begin{bmatrix} 1 & 0 & Tx \\ 0 & 1 & Ty \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} Sx & 0 & 0 \\ 0 & Sy & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} Sx & 0 & Tx \\ 0 & Sy & Ty \\ 0 & 0 & 1 \end{bmatrix}$$

* À noter que l'ordre du produit est l'inverse des transformations à effectuer. On souhaite obtenir un changement d'échelle puis une translation, on note donc TxS.

La combinaison est donc différent si on souhaite effectuer cette fois une translation suivie d'un changement d'échelle.

$$T(Tx,Ty) \times S(Sx, Sy) = \begin{bmatrix} Sx & 0 & 0 \\ 0 & Sy & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & Tx \\ 0 & 1 & Ty \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} Sx & 0 & Sx.Tx \\ 0 & Sy & Sy.Ty \\ 0 & 0 & 1 \end{bmatrix}$$

On peut ainsi exprimer la matrice de transformations composée des différentes transformations comme suit :

$$\begin{bmatrix} a & b & Tx \\ c & d & Ty \\ 0 & 0 & 1 \end{bmatrix}$$

avec a, b, c et d les éléments correspondant à la rotation, au changement d'échelle et au cisaillement.
et Tx, Ty, les éléments relatifs à la translation.

Jusqu'à présent, nous avons expliqué le fonctionnement des transformations géométriques dans un espace 2D. Nous allons maintenant voir le fonctionnement dans un espace 3D.

TRANSFORMATIONS 3D

Le principe dans un espace 3D est très similaire, c'est pourquoi cette partie n'est qu'un complément à tout ce qu'on vient de voir. Tout d'abord, on introduit aux vecteurs une composante supplémentaire z. En effet, un point peut maintenant se déplacer selon trois dimensions. Ainsi, en coordonnées homogènes, un point 3D est un vecteur à 4 coordonnées tel que :

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

En découle donc de nouvelles matrices de transformations très semblables à celles définies pour un espace 2D. Seule la matrice de rotation est un peu plus complexe, nous allons voir pourquoi.

- Translation

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & Tx \\ 0 & 1 & 0 & Ty \\ 0 & 0 & 1 & Tz \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

On obtient ainsi :

x' = x.1 + y.0 + z.0 + 1.Tx	y' = x.0 + y.1 + z.0 + 1.Ty	z' = x.0 + y.0 + z.1 + 1.Tz
x' = x + Tx	**y' = y + Ty**	**z' = z + Tz**

-> On obtient donc les expressions relatives à la translation dans un espace 3D.

- Changement d'échelle

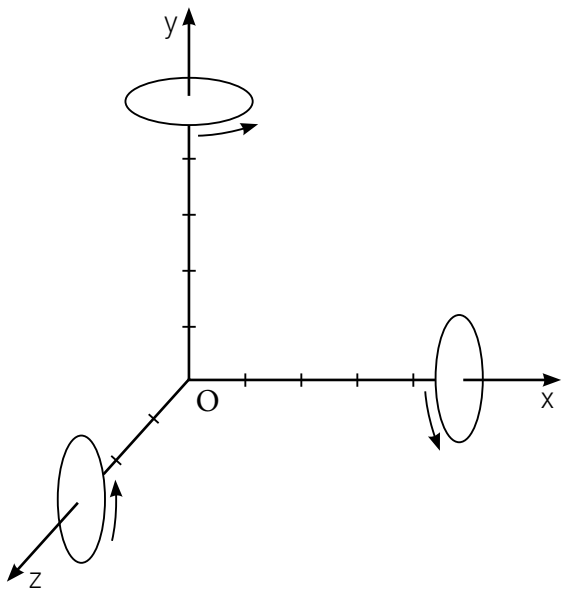
$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sy & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} x \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

On obtient ainsi :

x' = x.Sx + y.0 + z.0 + 1.0	y' = x.0 + y.Sy + z.0 + 1.0	z' = x.0 + y.0 + z.Sz + 1.0
x' = x.Sx	**y' = y.Sy**	**z' = z.Sz**

-> On obtient donc les expressions relatives au changement d'échelle dans un espace 3D.

Comme nous l'évoquions plus haut, la matrice de rotation est un peu plus complexe pour la simple raison que dans un espace 2D, la rotation ne se fait que selon un angle. Dans un espace 3D, la rotation se fait selon un angle mais aussi suivant un axe. La rotation d'un objet peut donc se faire autour de l'axe x, y ou z.



Rotation dans un espace 3D

On peut donc définir 3 matrices de rotation selon l'axe qui en dépend.

- Rotation sur l'axe z

Nous commencerons volontairement par l'axe z puisque la matrice de rotation en z dans un espace 3D est très proche de celle dans un espace 2D. En effet, à l'aide du schéma précédent, on s'aperçoit qu'une rotation autour de l'axe z fait pivoter l'objet de la même façon que dans un espace 2D, c'est à dire qu'une rotation en z tourne l'axe x vers l'axe y.

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} x \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

On obtient ainsi :

$x' = x.\cos(\theta) - y.\sin(\theta) + z.0 + 1.0$
$x' = x.\cos(\theta) - y.\sin(\theta)$

$y' = x.\sin(\theta) + y.\cos(\theta) + z.0 + 1.0$
$y' = x.\sin(\theta) + y.\cos(\theta)$

$z' = x.0 + y.0 + z.1 + 1.0$
$z' = z$

-> On obtient donc les expressions relatives à la rotation en z dans un espace 3D.

- Rotation sur l'axe x

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) & 0 \\ 0 & \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} x \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

On obtient ainsi :

$x' = x.1 - y.0 + z.0 + 1.0$
$x' = x$

$y' = x.0 + y.\cos(\theta) - z.\sin(\theta) + 1.0$
$y' = y.\cos(\theta) - z.\sin(\theta)$

$z' = -x.0 + y.\sin(\theta) + z.\cos(\theta) + 1.0$
$z' = y.\sin(\theta) + z.\cos(\theta)$

-> On obtient donc les expressions relatives à la rotation en x dans un espace 3D.

- Rotation sur l'axe y

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

On obtient ainsi :

$x' = x.\cos(\theta) - y.0 + z.\sin(\theta) + 1.0$
$x' = x.\cos(\theta) + z.\sin(\theta)$

$y' = x.0 + y.1 + z.0 + 1.0$
$y' = y$

$z' = -x.\sin(\theta) + y.0 + z.\cos(\theta) + 1.0$
$z' = -x.\sin(\theta) + z.\cos(\theta)$

-> On obtient donc les expressions relatives à la rotation en y dans un espace 3D.

Enfin, pour terminer cette partie «Transformations 3D», tout comme dans l'espace 2D, il est possible de combiner les transformations. On peut ainsi définir la matrice de transformations générale telle que :

$$\begin{bmatrix} a & b & c & Tx \\ d & e & f & Ty \\ g & h & i & Tz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

avec a, b, c, d, e, f, g, h et i les éléments correspondant à la rotation, au changement d'échelle et au cisaillement.
et Tx, Ty, Tz les éléments relatifs à la translation.

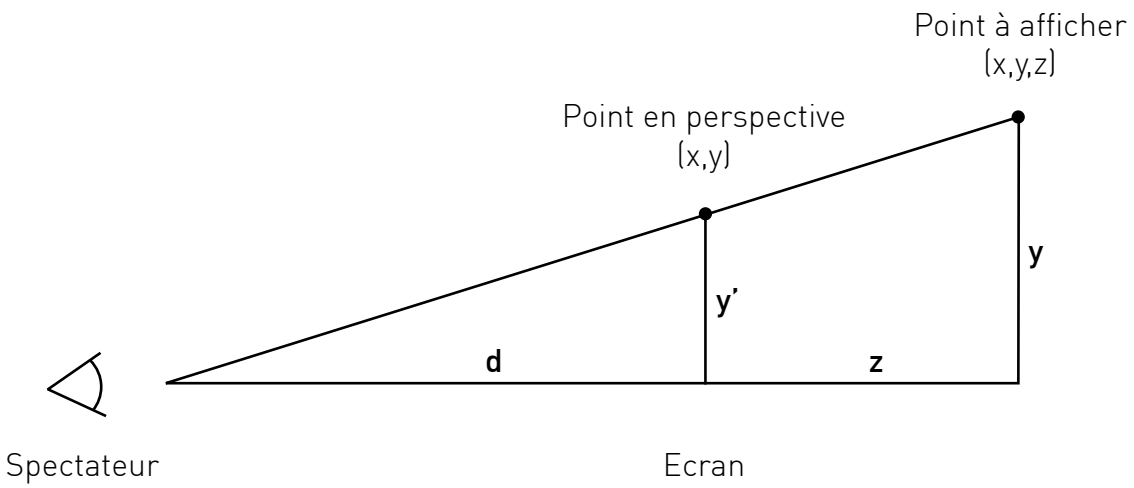
Nous pourrons rajouter un dernier point. Nous avons vu avec la rotation ou le changement d'échelle autour d'un point fixe, que souvent nous souhaitons déplacer un objet puis le remettre à sa position initiale. On utilise pour ça une matrice inverse, qui à la même forme que la matrice initiale mais on peut noter les relations suivantes :

- Tx etTy => -Tx et-Ty
- θ => -θ
- Sx et Sy => 1/Sx et1/Sy
- Cx et Cy => -Cx et -Cy

Pour terminer cette partie théorique sur les transformations géométriques, nous allons voir comment rendre un objet ou un environnement 3D sur un espace 2D tel qu'un écran de télévision par exemple.

PROJECTION PERSEPECTIVE

La projection consiste à afficher une scène 3D sur un plan. Et sur ce plan, c'est la perspective qui va donner une impression de 3D. On parle donc de projection de type perspective. Voyons le principe à l'aide d'un schéma :



Sur ce schéma, le principe consiste à projeter le point en 3D sur le plan de l'écran. Pour calculer les deux coordonnées (x,y) de ce point, on remarque qu'il est nécessaire d'utiliser le théorème de Thalès. Ainsi, on peut exprimer les relations suivantes :

$$\frac{d}{y'} = \frac{d+z}{y} \quad \text{d'où} \quad y' = \frac{d.y}{d+z}$$

On note $ratio = \frac{d}{d+z}$

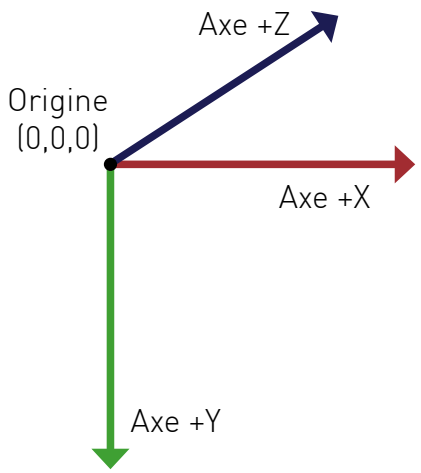
donc $y' = y.ratio$

$$\frac{d}{x'} = \frac{d+z}{x} \quad \text{d'où} \quad x' = \frac{d.x}{d+z}$$

donc $x' = x.ratio$

De cette façon, il suffit de projeter chaque point en 3D sur le plan de l'écran. Ce type de projection permet d'obtenir un effet visuel semblable à celui perçu par l'oeil humain. Ainsi, un objet éloigné sera plus petit que s'il était près par exemple.

Maintenant que nous avons abordé de nombreuses notions, nous allons pouvoir les mettre en pratique. Dans un premier temps nous verrons comment appliquer toutes ces notions en AS3, avec le logiciel Flash. Puis nous verrons quelques notions dans un autre langage, Processing, basé sur la plate-forme Java, et principalement destiné à la création graphique. À noter qu'avec Flash, le repère est un peu différent que celui qu'on à vu tout au long de ce compte rendu. L'axe y est orienté vers le bas et l'axe z correspond à la profondeur de l'écran. Le repère est orienté de cette façon puisqu'il se trouve dans le coin haut-gauche de l'écran.



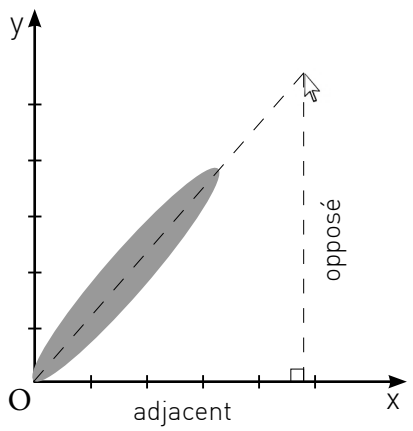
Repère dans Adobe Flash

APPLICATION EN AS3

TP 1 - TRIGONOMÉTRIE, PARTIE 1

Ce premier TP comporte 4 exercices permettant de mettre en pratique les notions de trigonométrie.

Exercice 1
Ce premier exercice consiste à orienter un segment en direction du pointeur de la souris. Pour ce faire, on considère un triangle rectangle dont le segment est l'hypoténuse. En se rapportant aux fonctions trigonométriques, on peut donc calculer l'angle à l'aide des coordonnées de la souris.



Dans un premier temps, on considère un triangle rectangle formé par le pointeur de la souris (cf. Schéma ci-contre). On peut donc facilement calculer le côté adjacent et le côté opposé de ce triangle rectangle.

```
// Calcul de la distance entre le pointeur de la souris et la coordonnée x du segment
// Correspond au côté adjacent en fonction de la souris
var dX:Number = mouseX-seg0.x;
// Calcul de la distance entre le pointeur de la souris et la coordonnée y du segment
// Correspond au côté opposé en fonction de la souris
var dY:Number = mouseY-seg0.y;
```

Ces calculs vont nous permettre de calculer un angle de rotation. En effet, à l'aide de la formule de la tangente, il est possible de trouver un angle en fonction des côtés adjacent et opposé.

```
// Calcul de l'angle en fonction des côté adjcent et opposé précédents
// En trigonométrie, on utilise la formule tan(angle)=oppposé/adjacent, puis on trouve la valeur de l'angle grace à Arctan
// En AS3, on utilise directement Math.atan2(oppposé, adjacent)
var angleRadian:Number = Math.atan2(dY, dX);
```

Il est ensuite nécessaire de convertir l'angle que nous venons de calculer. En effet, la fonction atan2 retourne un angle en radian. Or, pour appliquer une rotation à un objet, il faut que cet angle soit en degré.

```
// L'angle précédemment calculé étant en radian, on utilise une formule permettant de convertir en degré
var angleDegre:Number = angleRadian*180/Math.PI;
```

Il ne reste plus qu'à appliquer cet angle de rotation à notre segment.

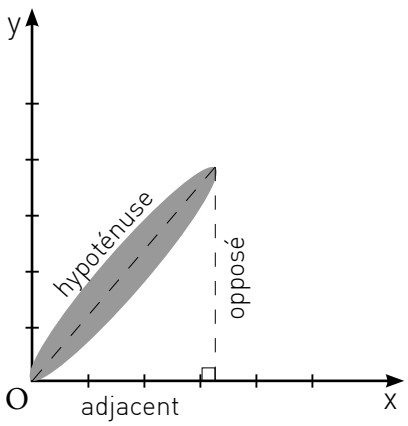
```
// Enfin, on applique une rotation au segment en fonction de cet angle calculé
// Ainsi, le segment va pivoter en suivant le pointeur de la souris
seg0.rotation = angleDegre;
```

Exercice 2

Le deuxième exercice est dans la continuité du premier. Il consiste toujours à orienter un segment en fonction du pointeur de la souris, mais cette fois, le segment n'est plus fixe mais se déplace également en fonction de la souris.

Dans un premier temps, on oriente donc le segment de la même façon que dans le premier exercice.

cf. Exercice 1



Puis dans un second temps, on déplace le segment en fonction de la souris en s'étant préalablement assurer de calculer le côté opposé et le côté adjacent en fonction de l'angle et de l'hypoténuse. Il s'agit donc des côtés adjacent et opposé du triangle rectangle formé par le segment. Ceci permet d'appliquer la position de la souris, non au point de pivot, mais à l'autre extrémité du segment.

```
// En considérant un triangle rectangle, le segment représente l'hypoténuse
// Puisqu'on connait l'angle, on va calculer le cote opposé en fonction de l'hypoténuse et de l'angle
// Soit opposé = hypoténuse * sin(angle)
var coteOppose = longSeg*Math.sin(angleRadian);
// On calcul également le côté adjacent, toujours en fonction de l'hypoténuse et de l'angle
// Soit adjacent = hypoténuse * cos(angle)
var coteAdjacent = longSeg*Math.cos(angleRadian);

// On va maintenant pouvoir déplacer le segment
// On le déplace en fonction du pointeur de la souris tout en conservant l'angle et en n'appliquant pas la position de la souris au point de pivot mais à l'autre extrémité
seg0.y = mouseY-coteOppose;
seg0.x = mouseX-coteAdjacent;
```

Exercice 3

Le troisième exercice de ce TP ne consiste plus seulement à orienter et déplacer un segment, mais deux. Un premier en fonction du pointeur de la souris, puis un second en fonction du premier segment.

Plutôt que de répéter plusieurs fois le code pour chacun des segments, nous allons écrire le code permettant d'orienter et de déplacer de façon abstraite. Pour cela, on utilise une fonction avec 3 paramètres, un pour les segments et deux autres pour les coordonnées. Ainsi, on peut écrire le code permettant d'orienter et de déplacer -le même que les exercices précédents- en utilisant ces paramètres qu'on renseignera par la suite.

```
function deplacer(seg:MovieClip, pX:Number, pY:Number) {
	// Déclaration des variables nécessaires
	var dX:Number;
	var dY:Number;
	var angleRadian:Number;
	var angleDegre:Number;

	// On calcul de manière abstraite la distance dX égale au paramètre pX moins la coordonnée x du paramètre seg
	// Correspond au côté adjacent
	dX = pX-seg.x;
	// On calcul de manière abstraite la distance dY égale au paramètre pY moins la coordonnée y du paramètre seg
	// Correspond au côté opposé
	dY = pY-seg.y;

	// Calcul de l'angle en fonction des côtés adjcent et opposé précédents
	// En trigonométrie, on utilise la formule tan(angle)=oppposé/adjacent, puis on trouve la valeur de l'angle grace à Arctan
	// En AS3, on utilise directement Math.atan2(oppposé, adjacent)
	angleRadian = Math.atan2(dY, dX);
	// L'angle précédemment calculé étant en radian, on utilise une formule permettant de convertir en degré
	angleDegre = angleRadian*180/Math.PI;
	// Enfin, on applique une rotation à l'élément passé en paramètre seg, en fonction de cet angle calculé
	// Ainsi, l'élément va pivoter en suivant le pointeur de la souris
	seg.rotation = angleDegre;
```

```
        // À ce stade, l'élément passé en paramètre seg ne se déplace pas
        // En considérant un triangle rectangle, le segment représente l'hypoténuse
        // Puisqu'on connait l'angle, on va calculer le cote opposé en fonction de
l'hypoténuse et de l'angle
        // Soit opposé = hypoténuse * sin(angle)
        var oppose = longSeg*Math.sin(angleRadian);
        // On calcul ensuite le côté adjacent, toujours en fonction de l'hypoténuse et
de l'angle
        // Soit adjacent = hypoténuse * cos(angle)
        var adjacent = longSeg*Math.cos(angleRadian);

        // On va maintenant pouvoir déplacer l'élément passé en paramètre seg
        // On le déplace en fonction des paramètres pX et pY, tout en conservant l'angle
        seg.x = pX-adjacent;
        seg.y = pY-oppose;
}
```

Il s'agit donc du même code que l'exercice 3. La différence est qu'on utilise plus directement les objets mais des paramètres qui seront renseigner. On va donc appeler une première fois cette fonction avec un premier segment en paramètre, orienter vers les coordonnées de la souris, qui sont les deux autres paramètres afin d'orienter et de déplacer le segment en fonction de la souris.

```
// Le segment seg0 est passé en paramètre seg, et les coordonnées x et y du pointeur
sont passées respectivement en paramètres pX et pY. Ainsi, pour cet appel de fonction,
on a :
// dx = mouseX-seg0.x et dy = mouseY-seg0.y puis seg0.rotation = angleDegre
// Enfin, seg0.x = mouseX-adjacent et seg0.y = mouseY-oppose;
// Donc, le segment seg0 s'oriente et se place sur le pointeur de la souris
deplacer(seg0, mouseX, mouseY);
```

Puis, on appelle une deuxième fois cette fonction[1], cette fois en passant le second segment en paramètre, avec les coordonnées du segment précédent pour les autres paramètres afin d'orienter et de déplacer le second segment en fonction le premier.

```
// Ici, le segment seg1 est passé en paramètre seg, et les coordonnées x et y de seg0
sont passées respectivement en paramètres pX et pY. Ainsi, pour cet appel de fonction,
on a :
// dx = seg0.x-seg1.x et dy = seg0.y-seg1.y puis seg1.rotation = angleDegre
// Enfin, seg1.x = seg.0-adjacent et seg1.y = seg0.y-oppose
// Donc, le segment seg1 s'oriente et se place à l'extrémité de seg0
deplacer(seg1, seg0.x, seg0.y);
```

[1] ces deux appels de fonction se font au sein d'une fonction lancer()

Exercice 4

Le quatrième et dernier exercice de ce TP n'est pas très éloigné de l'exercice précédent. En effet, il consiste cette fois à orienter et déplacer 6 segments. Le premier s'oriente et suit le pointeur de la souris, et tous les autres suivent et se déplacent en fonction du précédent. Ainsi, le deuxième s'oriente et vient se placer à l'extrémité du premier, le troisième s'oriente et vient se placer à l'extrémité du deuxième et ainsi de suite. La seconde différence est que les segments sont cette fois créés par le code.

C'est d'ailleurs par cette étape que l'on va commencer. À l'aide d'une boucle, nous allons pouvoir créer des segments, les ajouter et les placer sur à la scène, leur donner un nom ainsi que changer leur échelle.

```
// Rapport de 0.5 qui sera appliqué pour une mise à l'échelle des segments
var rapport:Number = 0.5;
// Nombre de segments à créer
var combien:uint = 6;
// On fait une boucle qui se répète 6 fois
for(var i=0;i<combien;i++) {
	// Création d'une variable seg de type Segment comme étant un nouveau Segment.
	var seg:Segment = new Segment();
	// Ajout des segments sur la scène
	addChild(seg);
	// On positionne les segments
	// Le premier segment se trouve en (30,30) car 30*(0+1) = 30
	// Puis les autres se place en fonction de i
	// Soit le deuxième segment en (60,60) car 30*(1+1) = 60 etc.
	seg.x = seg.y = 30*(i+1);
	// On nomme les segments créé, par «seg» suivi de la valeur de i
	// Soit le premier segment nommé seg0, le deuxième seg1 etc.
	seg.name = «seg»+i;
	// On change l'échelle des segments sur X et Y de 0.5, soit deux fois moins grand
	// Onchange l'échelle sur X et Y du même rapport, c'est donc une homotéthie
	seg.scaleX = seg.scaleY = rapport;
}
```

Pour le reste de cet exercice, il suffit de reprendre la fonction déplacer de l'exercice précédent avec les mêmes paramètres, afin de préparer l'orientation et le déplacement des segments. Ensuite, nous allons voir qu'il y a un léger changement lors de l'appel de ces fonctions.

On appelle toujours deux fois la fonction *deplace*. Mais cette fois, on ne peut plus appeler directement un segment en paramètre puisqu'il est créé par le code et possède un nom. On va donc utiliser une methode permettant de cibler un objet suivant son nom : getChildByName. On appelle donc une première fois la fonction *deplace* avec un premier segment en premier paramètre, ainsi que les coordonnées de la souris pour les deux autres paramètres afin d'orienter et de déplacer le segment en fonction de la souris ; exactement comme dans l'exercice 3.

```
// Le segment seg0 (appelé par la fonction getChildByNAme) est passé en paramètre seg,
et les coordonnées x et y du pointeur sont passé respectivement en paramètres pX et pY
// Ainsi, pour cet appel de fonction, on a :
// dx = mouseX-seg0.x et dy = mouseY-seg0.y
// seg0.rotation = angleDegre
// Enfin, seg0.x = mouseX-adjacent et seg0.y = mouseY-oppose;
// Donc, le segment seg0 s'oriente et se place sur le pointeur de la souris
deplace(getChildByName(«seg0»), mouseX, mouseY);
```

Puis, on appelle une deuxième fois cette fonction *deplace*, cette fois en passant tous les autres segments en premier paramètre -toujours en les apellant à l'aide de la méthode getChildByNAme- avec les coordonnées de tous les segment précédents pour les deux autres paramètres afin d'orienter et de déplacer chaque segment, à chaque fois en fonction du segment précédent. On utilise donc une boucle pour pouvoir appeler tous les segments.

```
// Ici on crée une boucle permettant de placer chaque segment à l'extrémité du segment
précédent
// toujours en appellant les segments à l'aide de la fonction getChildByName
for(i=1; i<combien; i++){
      // Ici, le segment à créé est passé en paramètre seg
      // et les coordonnées x et y du segment précédent sont passées respectivement en
paramètres pX et pY. Ainsi, pour cet appel de fonction, on a :
      // dx = seg0.x-seg1.x et dy = seg0.y-seg1.y pour i=1
      // Puis dx = seg1.x-seg2.x et dy = seg1.y-seg2.y pour i=2 etc.
      // De la même manière, les autres calculs de la fonction deplace() sont effectués
      // seg1.rotation = angleDegre pour i=1, seg2.rotation = angleDegre pour i=2 etc.
      // Enfin, seg1.x = seg.0-adjacent et seg1.y = seg0.y-oppose pour i=1 etc.
      // Donc, le segment seg1 s'oriente et se place à l'extrémité de seg0 et ainsi de
suite
      deplace(getChildByName(«seg»+i),getChildByName(«seg»+(i-1)).x,
getChildByName(«seg»+(i-1)).y );
}
```

TP 2 - TRIGONOMÉTRIE, PARTIE 2

Ce deuxième TP comporte 3 exercices (les 6, 7 et 8) permettant de continuer à mettre en pratique les notions de trigonométrie.

Exercice 6

Ce premier exercice du deuxième TP consacré aux fonctions trigonométriques reprend des notions déjà vu, dans l'exercice 3 notamment. Il consiste à orienter un segment en direction du pointeur de la souris puis un second en fonction du premier, tout en restant fixe.

Comme pour les premiers exercice, on oriente un premier segment vers le pointeur de la souris.

```
// Orientation de seg0 vers la position du pointeur de la souris
// Calcul de la distance entre le pointeur de la souris et la coordonnée x du segment seg0
// Correspond au côté adjacent du triangle rectangle formé par la souris
var dX0:Number = mouseX-seg0.x;
// Calcul de la distance entre le pointeur de la souris et la coordonnée y du segment seg0
// Correspond au côté opposé du triangle rectangle formé par la souris
var dY0:Number = mouseY-seg0.y;

// Calcul de l'angle en fonction des côté adjcent et opposé précédents
// En trigonométrie, on utilise la formule tan(angle)=oppposé/adjacent, puis on trouve la valeur de l'angle grace à Arctan
// En AS3, on utilise directement Math.atan2(oppposé, adjacent)
var angleRadian0:Number = Math.atan2(dY0, dX0);
// L'angle précédemment calculé étant en radian, on utilise une formule permettant de convertir en degré
var angleDegre0:Number = angleRadian0*180/Math.PI;
// Enfin, on applique une rotation au segment en fonction de cet angle calculé
// Ainsi, le segment va pivoter en suivant le pointeur de la souris
seg0.rotation = angleDegre0;
```

L'étape principale est que l'on ne déplace pas seg0, mais on stocke ses positions temporairement dans des variables. On va ainsi pouvoir orienter seg1 vers seg0.

```
// En considérant un triangle rectangle, le segment représente l'hypoténuse
// Puisqu'on connait l'angle, on va calculer, pour seg0, le cote opposé en fonction de l'hypoténuse et de l'angle
// Soit opposé = hypoténuse * sin(angle)
var coteOppose0 = longSeg*Math.sin(angleRadian0);
// On calcul également le côté adjacent, toujours en fonction de l'hypoténuse et de l'angle
// Soit adjacent = hypoténuse * cos(angle)
var coteAdjacent0 = longSeg*Math.cos(angleRadian0);

// Calcul de la position de seg0
// On ne déplace pas seg0 mais on stocke ses positions temporairement
var tempX:int = mouseX-coteAdjacent0;
var tempY:int = mouseY-coteOppose0;

// Orientation de seg1 vers les position temporaires calculées précédemment
// Calcul de la distance entre tempX et la coordonnée x du segment seg1
// Correspond au côté adjacent
var dX1:Number = tempX-seg1.x;
// Calcul de la distance entre tempY et la coordonnée y du segment seg1
// Correspond au côté opposé
var dY1:Number = tempY-seg1.y;

// De même que précédemment, on calcul l'angle en fonction du côté adjcent et du côté opposé
var angleRadian1:Number = Math.atan2(dY1, dX1);
// L'angle précédemment calculé étant en radian, on utilise une formule permettant de convertir en degré
var angleDegre1:Number = angleRadian1*180/Math.PI;
// Enfin, on applique une rotation au segment en fonction de cet angle calculé
// Ainsi, le segment seg1 va s'orienter vers seg0
seg1.rotation = angleDegre1;
```

Il ne reste plus qu'à déplacer seg0 à l'extrémité de seg1.

```
// On déplace seg0 à l'extrémité de seg1
seg0.x = seg1.x + (longSeg*Math.cos(angleRadian1));
seg0.y = seg1.y + (longSeg*Math.sin(angleRadian1));
```

Exercice 7

Cet exercice consiste à combiner l'exercice précédent, avec les passages de paramètres que l'on a vu dans l'exercice 3. Il s'agit d'obtenir 6 segments cette fois, avec toujours un qui s'oriente vers le pointeur de la souris, les autres qui s'orientent et se placent entre eux au fur et à mesure, et enfin, un premier segment fixe.

Comme pour chaque exercice, on a une fonction qui nous permet de préparer l'orientation de chacun des segments. Dans cet exercice, on va de nouveau se servir d'une fonction avec toujours trois paramètres, un pour le segment, et deux autres pour les coordonnées. Simplement cette fois, on ne déplace pas les segments mais on stocke les positions dans des variables, puis on retourne un point avec ces coordonnées.

```
// Fonction permettant d'orienter les segments
function chercher(seg:MovieClip, posX:Number, posY:Number):Object {
	// On calcul de manière abstraite la distance dX égale au paramètre posX moins la coordonnée x du paramètre seg
	var dX:Number = posX-seg.x;
	// On calcul de manière abstraite la distance dY égale au paramètre posY moins la coordonnée y du paramètre seg
	var dY:Number = posY-seg.y;

	// Calcul de l'angle en fonction des côtés adjcent et opposé précédents
	// En trigonométrie, on utilise la formule tan(angle)=oppposé/adjacent, puis on trouve la valeur de l'angle grace à Arctan
	// En AS3, on utilise directement Math.atan2(oppposé, adjacent)
	var angleRadian:Number=Math.atan2(dY,dX);
	// L'angle précédemment calculé étant en radian, on utilise une formule permettant de convertir en degré
	var angleDegre:Number=180*angleRadian/Math.PI;
	// Enfin, on applique une rotation à l'élément passé en paramètre seg, en fonction de cet angle calculé
	seg.rotation = angleDegre;
```

```
	// En considérant un triangle rectangle, le segment représente l'hypoténuse
	// Puisqu'on connait l'angle, on va calculer le cote opposé en fonction de l'hypoténuse -le segment- et l'angle
	// Soit opposé = hypoténuse * sin(angle)
	var coteOppose = longSeg*Math.sin(angleRadian);
	// On calcul également le côté adjacent
	// Soit adjacent = hypoténuse * cos(angle)
	var coteAdjacent = longSeg*Math.cos(angleRadian);

	// On ne déplace pas le segment mais on stock les positions
	// toujours en fonction du pointeur de la souris tout en conservant l'angle
	// et en n'appliquant pas la position de la souris au point de pivot mais à l'autre extrémité du segment
	var tempX:Number = posX-coteAdjacent;
	var tempY:Number = posY-coteOppose;

	// Le Point p prend en paramètre les coordonnées temporaires précédentes
	var p:Point = new Point (tempX,tempY);
	// Renvoi les valeurs du Point p
	return p;
}
```

Une fois l'orientation des segments préparée, on va pouvoir préparer le déplcement de ces segments. Dans l'exercice précédent, nous avons placé seg0 au bout de seg1, cette fois on va tout simplement faire la même chose mais avec des paramètres afin de placer segA au bout de segB. Nous aurons besoin de recalculer l'angle de segB en radian afin de calculer la position du bout de ce segment.

```
// Fonction permettant de positionner les segments entre eux
function position(segA:MovieClip, segB:MovieClip):void {
	// On récupère l'angle de segB en degré, puis on le converti en radian
    var angleRadian:Number = (segB.rotation*Math.PI)/180;
	// On place l'élément passé en paramètre segA à l'extrémité de l'élément passé en paramètre segB
	segA.x = segB.x+longSeg*Math.cos(angleRadian);
	segA.y = segB.y+longSeg*Math.sin(angleRadian);
}
```

Maintenant que l'orientation et le déplacement des segments sont prêt, nous allons pouvoir appeler ces fonctions en passant les segments et les coordonnées en paramètres.On oriente donc un premier segment vers le pointeur de la souris, ensuite on oriente chaque segment par rapport au segment précédent, puis on place chaque segment à l'extrémité du segment précédent là encore.

```
// Fonction permettant d'orienter et de placer les segments
function deplacer(pEvt:Event):void {
	// Avant de lire ces commentaires, se référer à la fonction chercher()
	// Appel de la fonction chercher() pour :
	// Orienter seg0 vers le pointeur de la souris
	// Chercher la position de seg0 par rapport à la position du pointeur de la souris
	var p1:Point = chercher(seg0, mouseX, mouseY);
	// Appel de la fonction chercher() dans une boucle pour :
	// Orienter chaque segment par rapport au segment précédent
	// Chercher la position temporaire de chaque segment par rapport au segment précédent
	for (i=1;i<numSegs;i++){
			p1 = chercher(this[«seg»+i],p1.x,p1.y);
	}
	// Avant de lire ces commentaires, se référer à la fonction position()
	// Appel de la fonction position() pour :
	// Placer chaque segment à l'extrémité du segment précédent
	for (i=numSegs-1;i>0;i--){
			position(this[«seg»+(i-1)],this[«seg»+i]);
	}
}
```

Exercice 8

Ce dernier exercice du TP est très similaire au précédent, il suffit simplement de rajouter une boule qui se déplace dans la scène, et orienter le premier segment vers cette boule et non plus vers le pointeur de la souris. On pourra également appliquer un changement de vitesse et de direction lorsqu'il y a collision avec le segment.

On prépare donc les limites de la scène ainsi que les variables de vitesse de la boule.

```
// Vitesse de déplacement de la boule en x
var vX:Number = 5;
// Vitesse de déplacement de la boule en y
var vY:Number = 5;
// Limite Haute de la scène
var limH:Number = boule_mc.height/2;
// Limite Basse de la scène
var limB:Number = stage.stageHeight-limH;
// Limite Gauche de la scène
var limG:Number = boule_mc.width/2;
// Limite Droite de la scène
var limD:Number = stage.stageWidth-limG;
```

En reprenant le code précédent, on oriente non plus le premier segment vers le pointeur de la souris mais vers la boule.

```
// Avant de lire ces commentaires, se référer à la fonction chercher()
// Appel de la fonction chercher() pour :
// Orienter seg0 vers la position de la boule
// Chercher la position de seg0 par rapport à la position de la boule
var p1:Point= chercher(seg0, boule_mc.x , boule_mc.y);
```

Il ne reste plus qu'à déplacer la boule dans la scène et créer un petit effet lorsqu'il y a collision avec le segment.

```
// Fonction qui permet de déplacer la boule
function deplacerBoule():void {
	// On stocke les positions en x et en y de la boule
	// Celle-ci se déplace en x à l'aide de la variable vX=5, et en y à l'aide de la variable vY=5
	// Autrement dit, la boule se déplace à une cadence de 5pixels
	var posBouleX:Number = boule_mc.x+vX;
	var posBouleY:Number = boule_mc.y+vY;
	// On va poser des conditions pour que la boule rebondisse lorsqu'elle atteint les limites Droite et Gauche
	// Si la position en x de la boule est inférieure à la limite Gauche
		if(posBouleX > limD) {
		// On inverse le sens de déplacement de la boule en x
		vX = -vX;
		}
	// Sinon, si la position en x de la boule est supérieure à la limite Droite
	else if(posBouleX < limG) {
		// On inverse le sens de déplacement de la boule en x
		vX = -vX;
		}

	// On va poser des conditions pour que la boule rebondisse lorsqu'elle atteint les limites Haute et Basse
	// Si la position en y de la boule est inférieure à la limite Haute
		else if(posBouleY < limH) {
		// On inverse le sens de déplacement de la boule en y
		vY = -vY;
		}
	// Sinon, si la position en x de la boule est supérieure à la limite Basse
	else if(posBouleY > limB) {
		// On inverse le sens de déplacement de la boule en y avec en plus un effet de rebond
		vY = -vY*bounce;
		}
		// On affecte ensuite les positions de la boule
		boule_mc.x = posBouleX;
		boule_mc.y = posBouleY;
}
```

La méthode hitTestObject permet de verifier s'il y a collision.

```
// Fonction qui permet de vérifier une collision
function verifierCollision():void {
	// Si la balle heurte seg0
	if (boule_mc.hitTestObject(seg0)){
		// Alors la vitesse en y décélère
		// À l'inverse, lorsque la boule ne sera plus en contact avec seg0, elle va se mettre à accélérer
			vY = vY-(2*accel);
	}
}
```

TP 3 - TRANSFORMATIONS GÉOMÉTRIQUES 2D

Ce troisième TP à pour but de mettre en pratique les transformations géométriques 2D. Nous utiliserons donc celles que l'on a vu dans ce compte rendu. Nous aborderons également les matrices.

Scène Bob et Bobette
Cet exercice consiste à appliquer les principales transformations géométriques à une image, à savoir, la translation, la rotation, le changement d'échelle ainsi que le cisaillement, le tout en modifiant les éléments correpondant dans la matrice.

Tout d'abord, les éléments de la matrice sont illustrés par des NumericSteppers. On reproduit donc la matrice de transformation générale dans l'espace 2D qu'on a pu voir :

$$\begin{bmatrix} a & b & Tx \\ c & d & Ty \\ 0 & 0 & 1 \end{bmatrix}$$

On va donc créer et ajouter les NumericsSteppers correspondant. La class Matrix contient les paramètres a, b, c, d, Tx et Ty qui correspondent aux éléments de la matrice.

```
// Élément A de la matrice
elmt_A = new NumericStepper();
// Valeur par défaut du NumericStepper
// Ici, l'élément a de notre matrice de transformation correspond par défaut à l'élément
a de la matrice de l'image copie
elmt_A.value = copie.transform.matrix.a;
```

De la même façon, on ajoute et initialise les éléments b, c, d, Tx et Ty.

Ensuite, nous allons mettre en surbrillance le bouton de transformation lorsqu'on clique dessus et activer les éléments de la matrice correspondant. Si je clique sur translation, les éléments Tx et Ty seulement doivent être actif.

```
// Activer les éléments correspondant à la translation
buttonTranslation.emphasized = true;
elmt_Tx.enabled = true;
elmt_Ty.enabled = true;
```

De cette façon, en modifiant la valeur des NumericSteppers, on change les valeurs de la matrice et donc les transformations s'appliquent à l'image.

Il reste à mettre en place la rotation, puisque pour cette transformation, les éléments a, b, c et d ne sont pas indépendants mais au contraire fonctionnent ensemble. Rappelons la matrice de rotation :

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} x \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

On peut donc renseigner les éléments a, b, c et d de la matrice comme la matrice de rotation ci-dessus. Il faut donc poser une condition et remplir ce 4 éléments ensemble lorsque que la rotation est sélectionner, puisque rappelons que le cisaillement et le chagement d'échelle utilise aussi quelques uns de ces éléments, à savoir a et d pour le changement d'échelle et b et c pour le cisaillement.

Il reste ensuite à appliquer les éléments correspondant à la matrice de rotation à la matrice de l'image. Ou alors, si la rotation n'est pas sélectionnée, d'appliquer les éléments indépendamment.

```
// Application de la matrice Rotation si le bouton Rotation est actif
if (buttonRotation.emphasized == true) {
	// Angle radian généré entre 0 et 2PI (360)
	var angleRadian:Number = Math.random()*(2*Math.PI);

	// Actualisation des NumericSteppers selon l'angle radian
	elmt_A.value = Math.cos(angleRadian);
	elmt_B.value = Math.sin(angleRadian);
	elmt_C.value = Math.sin(-angleRadian);
	elmt_D.value = Math.cos(angleRadian);

	// Application de la matrice de rotation selon l'angle radian
	var matrixRotation:Matrix = new Matrix(Math.cos(angleRadian), Math.
sin(angleRadian), Math.sin(-angleRadian), Math.cos(angleRadian), elmt_Tx.value, elmt_
Ty.value);
	this.copie.transform.matrix = matrixRotation;
}
```

```
// Sinon, actualisation de la matrice de l'image selon les valeurs des NumericSteppers,
indépendamment les uns des autres
else {
      // Application de cette matrice de transformation (celle de la scène)
      var matrixChange:Matrix = new Matrix(elmt_A.value, elmt_B.value, elmt_C.value,
elmt_D.value, elmt_Tx.value, elmt_Ty.value);
      this.copie.transform.matrix = matrixChange;
}
```

Le dernier point de ce TP consiste à déplacer un point, Bob, sur l'image d'origine. Un point présent sur l'image copie, Bobette, doit se déplacer de la même façon, mais ce point doit également subir les transformations de l'image copie.

Nous déplacerons le point Bob de la même façon que nous avons déplacer la boule de l'exercice 8 de trigonométrie. Seulement là il ne se déplacera pas automatiquement, mais uniquement en utilisant les flèches de l'interface. On calcul donc les limites de l'image comme précédemment et on peut déplacer notre point de cette manière :

```
// Si bob ne dépasse pas la limite gauche
if (bob.x > limG) {
// bob se déplace de 4 pixels vers la gauche
bob.x = bob.x-4;
```

De cette manière, on déplace notre point vers la droite, vers le haut et vers le bas.

Pour que notre point dans l'image copie, Bobette se déplace de la même façon que Bob dans son image, il faut cloner la matrice d'origine de Bob puis l'appliquer à Bobette. Si nous appliquons directement la matrice de Bob à Bobette sans cloner, Bobette se retrouve superposée à Bob.

```
// On applique la matrice de bob à bobette
// Ceci fonctionne, car bobette est placée dans l'image copie
bobette.transform.matrix = bob.transform.matrix.clone();
```

* Comme il est mentionné dans le commentaire, ici, Bobette se déplace de la même façon que Bob et subit les transformations de l'image copie CAR Bobette est placé directement dans l'image copie. Nous allons voir comment obtenir le même résultat lorsque Bobette est placée sur la scène.

Pour que Bobette se déplace de la même façon que Bob et subisse les transformations de l'image, il va falloir, comme nous l'avons vu dans le compte rendu, combiner les matrices de Bob et celle de l'image. J'ai donc ajouté un troisième point, Bobus, qui réagit de la même façon que Bobette, seulement Bobus est placé non pas dans l'image copie mais dans la scène.

En AS3, la méthode concat permet de combiner deux matrices.

```
// tempBob devient la matrice concaténée de tempBob et tempCopie
tempBob.concat(tempCopie);
// Contrairement à bobette, bobus est placé sur la scène, donc il faut lui appliquer les transformations de bob mais aussi de l'image copie
// On applique la matrice concaténée à bobus
bobus.transform.matrix = tempBob;
```

Ici, tempBob.contat(tempCopie) transforme la matrice tempBob en une matrice qui applique les transformations de la matrice de Bob puis celle de la matrice de l'image copie.

TP 4 - PROJECTION PERSPECTIVE

Ce quatrième TP vise à mettre en pratique le principe de projection perspective que l'on a abordé en toute fin de ce compte rendu.

Lettre en perspective

Cet exercice va permettre de créer des lettres en perspective, ce qui va donner une impression de 3D.

Nous commençons par créer un point et calculer sa projection. Nous avons vu dans le cours comment obtenir un point 2D à partir d'un point 3D :

$$x' = \frac{d.x}{d+z} \qquad y' = \frac{d.y}{d+z}$$

- Classe Point3D

```
public function projection():Point {
	// Création d'un Point
	var monPoint:Point = new Point();
	// Calcul des coordonnées x et y de monPoint.
	// Calcul de la projection avec la formule x'=d.x/d+z
	monPoint.x = posX * Main.DIST/(Main.DIST + posZ);
	// Calcul de la projection avec la formule y'=d.y/d+z
	monPoint.y = posY * Main.DIST/(Main.DIST + posZ);
	// Retourne les valeurs de monPoint
	return monPoint;
}
```

Ensuite, nous allons préparé un tableau qui va permettre de stocker les points dont nous allons avoir besoin pour créer nos lettres, puis nous allons faire en sorte que chaque point soit relié. En AS3, moveTo(x,y) permet de se placer simplement au coordonnées x et y. Puis à l'aide de lineTo(x,y), nous pouvons relié le point renseigné dans moveTo au point renseigné dans lineTo.

- Classe Forme

```
// Fonction permettant de relier les points entre eux
public function reliePoints(tabPoints:Array){
	// On crée un premier point que l'on met dans le tableau
	var p0:Point = tabPoints[0].projection();
	var i:int=1;

	// On se positionne aux coordonnées p0.X, p0.y à l'aide de moveTo()
	this.graphics.moveTo(p0.x, p0.y);

	// On crée une boucle qui va s'éxécuter pour chaque point
	for (i=1; i<tabPoints.length; i++) {
		// On crée les autres points nécessaires, que l'on met aussi dans le
tableau
		var p:Point = tabPoints[i].projection();
		// On relie les points entre eux à l'aide de lineTo()
		this.graphics.lineTo(p.x,p.y);
	}
	// On termine en reliant le dernier point au premier
	this.graphics.lineTo(p0.x,p0.y);
}
```

Nous allons maintenant créer deux points 3D, un qui va correspondre à la forme avant, et un correspondant à la forme arrière.

- Classe Solide

```
public function ajouteTraces(tabPoints:Array, interieur:Boolean = false, indice:int =
0) {
	var i:int = 0;
	// Pour chaque points 2D créés
	for (i=0; i<tabPoints.length; i++) {
		// On crée un point 3D en lui attribuant une coordonnée Z pour la face
avant et la face arrière
		var p1:Point3D = new Point3D(tabPoints[i].x, tabPoints[i].y, 0.8);
		var p2:Point3D = new Point3D(tabPoints[i].x, tabPoints[i].y, -1);

		// On ajoute ensuite les 2 points
		tabFormes[0].ajouteSommetTrace(p1, interieur, indice);
		tabFormes[1].ajouteSommetTrace(p2, interieur, indice);
	}
}
```

Maintenant que la projection et les points sont prêts, nous allons pouvoir créer nos lettres en ajoutant les coordonnées de nos points. Nous ajoutons nos coordonnées en fonction d'un point de projection. Ainsi, pour créer une lettre K, nous procéderons comme suit :

- Classe Main

1. On crée notre point

```
// Initiale K
public var point:Solide;
```

2. On le définie comme un nouveau solide puis on l'ajoute sur la scène

```
// On crée un nouveau solide vide
point = new Solide();
// Création d'un solide à la forme de l'initiale K
creationK(point);
// Ajout du solide dans la liste d'affichage de la scène
this.addChild(point);
```

3. On positionne notre lettre dans la scène

```
// On place notre solide dans la scène
point.x = this.stage.stageWidth/4;
point.y = this.stage.stageHeight/1.5;
```

4. On ajoute les coordonnées de notre points

```
// fonction permettant de dessiner la lettre
public function creationK(initialeK:Solide):void{
	// Création du tableau de sommets extérieurs
	var tabPoints:Array = new Array();

	tabPoints.push(new Point(2,  -6));
	tabPoints.push(new Point(3,  -6));
	tabPoints.push(new Point(3,  -4));
	tabPoints.push(new Point(4,  -6));
	tabPoints.push(new Point(5,  -6));
	tabPoints.push(new Point(7/2,-3));
	tabPoints.push(new Point(5,   0));
	tabPoints.push(new Point(4,   0));
	tabPoints.push(new Point(3,  -2));
	tabPoints.push(new Point(3,   0));
	tabPoints.push(new Point(2,   0));

	initialeK.creeSolideExtrude(tabPoints);
}
```

TP 5 - TRANSFORMATIONS GÉOMÉTRIQUES 3D

Ce cinquième et dernier TP vient clore la mise en pratique des transformations géométriques. Dans ce TP nous allons aborder les transformations 3D.

Galerie cylindrique 3D

Cet exercice consiste à créer une galerie d'images sous la forme d'un cylindre de 3 lignes et 30 colonnes, le tout en exploitant les fonctions 3D natives que propose Flash.

Dans un premier temps, nous préparons toutes les variables qui vont nous servir pour créer notre cylindre.

```
// Conteneur
var conteneur:Sprite = new Sprite();
// Tableau pour les vignettes
var tabVignettes:Array = new Array();
// Grande image
var imageGrande:Sprite = new Sprite();
// Nombre de vignettes
var nbVignettes:int = 90;
// Largeur d'une vignette
var largeurVignette:int = 110;
// Hauteur d'une vignette
var hauteurVignette:int = 80;
// Espacement vertical entre les vignettes
var espaceVertical:int = 15;
// Nombre de lignes
var nbLignes:int = 3;
// Nombre de colonnes
var nbColonnes:int = 30;
// Rayon du cylindre
var rayonCylindre:int = 600;
```

On ajoute puis positionne ensuite le conteneur

```
addChild(conteneur);
conteneur.x = stage.stageWidth / 2;
conteneur.y = stage.stageHeight / 2;
conteneur.z = -600;
```

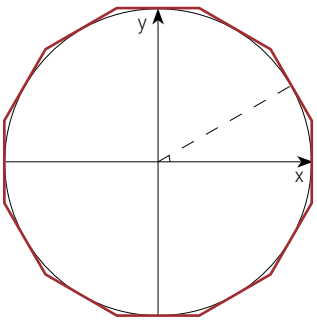
Il reste maintenant à créer et ajouter les vignettes, à savoir 90, puis les disposer afin de former un cylindre.

```
for (var i:int = 0; i<nbVignettes; i++) {
	// Création de vignette de type Sprite
	var vignette:Sprite = new Sprite ();
	// Indice Ligne
	var numLigne = Math.floor(i/nbColonnes);
	// Indice Colonne
	var numColonne = i%nbColonnes;
	// Angle servant à disposer les vignettes
	var angleVignette:Number = numColonne*((2*Math.PI)/nbColonnes);

	// Position de la vignette en X
	vignette.x = Math.sin(angleVignette)*rayonCylindre;
	// Position de la vignette en Y
	vignette.y = (numLigne-1)*(espaceVertical+hauteurVignette);
	// Position de la vignette en Z
	vignette.z = Math.cos(angleVignette)*rayonCylindre;

	// Orientation des vignettes
	vignette.rotationY = angleVignette*180/Math.PI;

	// Ajoute la vignette dans le conteneur
	conteneur.addChild(vignette);
	// Ajoute la vignette créée dans le tableau
	tabVignettes.push(vignette);
}
```



Pour placer les vignettes, on peut s'aider du cercle trigonométrique

Pour terminer ce compte rendu, nous allons mettre en pratique quelques unes des notions que nous avons vu mais avec un autre langage, Processing.

PROCESSING

GÉNÉRALITÉ

Processing est un langage de programmation principalement destiné à la création plastique et graphique interactive. Le logiciel du même nom est disponible sur de nombreux système d'exploitation comme Macintosh, Windows, Linux mais aussi Android puisqu'il est basé sur la plate-forme Java.

Nous avons mis en pratique des notions avec Processing parce qu'il est assez simple à prendre en main surtout quand on est passé par l'ActionScript. D'ailleurs, voyons quelques fonctions de base très pratiques :

size(l,h); : taille de l'affichage, de largeur l et hauteur h
rect(x,y,l,h); : dessine un rectangle de coordonnés x,y et de largeur l et hauteur h
fill(128,128,128); : remplissage en couleur r, v et b
frameRate : renvoie le nombre d'images par seconde

void setup() {...} : fonction éxécutée une seule fois, à l'ouverture de processing
void draw(){...} : fonction éxécutée tout le temps, comme ENTER_FRAME sur Flash

Interface de Processing

EXEMPLES DE PROGRAMMES

CUBE AU COULEURS DU WEB - Solution 1

Voyons un premier programme consistant à réaliser un cube constitué de 6x6x6 cubes représentant les couleurs du web. Nous verrons aussi comment optimiser ce code pour qu'il fonctionne toujours en changeant le nombre de cubes.

L'idée est de créer nos cubes à l'aide de 3 boucles, une pour les lignes, une pour les colonnes et une pour la profondeur. Pour le remplissage des cubes en couleurs, nous ne pouvons pas représenter les 255 niveaux de couleurs puisqu'on a pas assez de cubes. On va donc laisser de côtés certaines nuances et on va remplir les cubes par intervalles. Ainsi, avec 6 cubes on peut définir :

5 x 51 = 255. On rempli donc les cubes à intervalle de 51 nuances.
* On utilise 5 car 6 cubes correspond à 6 élément de 0 à 5
Donc le premier cube aura une composante de couleur à 0, le deuxième à 51, le troisième à 102 et ainsi de suite jusqu'à 255.

Plutôt que de l'écrire directement, on peut stocker cette constante permettant de définir l'intervalle dans une variable afin d'optimiser le programme :

cstCouleur = 255/nbCubes-1; avec nbCubes la variable définissant le nombre de cubes

Ainsi, en changeant le nombre de cubes, la constante permettant de calculer l'intervalle se met à jour. Voyons maintenant concrêtement ce que donne le code.

Tout d'abord, on définie les variables dont on va avoir besoin :

```
// Taille du côté d'un cube
int cote;
// Espacement entre les cubes
int pas;
// Translation effectuée
int trans;
// Nombre de cubes à générer
int nbCubes;
// Pas permettant de remplir les cubes
int cstCouleur;
```

Ensuite, on définie la méthode setup() en fixant les paramètres de la scène et en affectant les différentes variables :

```
void setup() {
  // Le côté d'un cube sera de 100 pixels
  cote = 100;
  // L'espacement entre les cubes est de 10 pixels
  pas = 10;
  // La translation a effectuée est égale au côté d'un cube + l'espacement
  // Ici, la translation est de 100+10 = 110 pixels
  trans = cote+pas;
  // Il y a 6 cubes à générer
  nbCubes = 6;
  // Pas servant à remplir les cubes, la constante
  cstCouleur = 255/nbCubes-1;
  // L'affichage à une taille de 800*800
  // Utilisation du moteur 3D de Processing
  size(800, 800, P3D);
  // Background de la scène gris
  // Il n'a qu'une seule valeur, c'est donc un gris
  // Il s'agit d'un gris moyen
  background(128);
  // Les cubes n'auront pas de contours
  noStroke();
}
```

Maintenant, on va pouvoir créer nos cubes et les placer de façon a obtenir un cube.

```
void draw() {
  // Le background est généré à chaque fois pour qu'à chaque mouvement, la scène soit
nettoyée
  // Le background généré est de la même couleur que le background initial
  background(128);
  // On prépare la matrice de transformation au préalable
  // Ici, on se place en x=10, y=10 et z=-1000 (L'élément final va être grand)
  translate(10, 10, -1000);
```

```
  // On applique une rotation en X
  // Lorsque la souris bouge vers le haut ou vers le bas, l'élément fera une rotation sur l'axe X
  rotateX(mouseY*0.01);
  // On applique une rotation en Y
   // Lorsque la souris bouge vers la gauche ou vers la droite, l'élément fera une rotation sur l'axe X
  rotateY(mouseX*0.01);

  // On va dessiner un cube représentant les couleurs du web
  // On va dessiner 6 cubes sur l'axe X grâce à la translation
  for (int i=0; i<nbCubes; i++) {
    // On va translater les 6 cubes précédents selon 6 lignes sur l'axe Y
    for (int j=0; j<nbCubes; j++) {
     // On va translater les 6 lignes de cubes précédentes selon 6 colonnes sur l'axe Z
      for (int k=0; k<nbCubes; k++) {
        // On enregistre la matrice de transformation courante
        pushMatrix();
        // On prépare la matrice de transformations au préalable
       // On translate à partir du premier cube, à chaque fois que la boucle s'éxécute
        // Au final, on translate donc 6 fois sur x, 6 fois sur y et 6 fois sur z
        // Au départ on translate d'une fois à partir du premier cube
         // Ensuite, on translate deux fois à partir du premier cube pour en dessiner un à côté du deuxième etc..
        translate(i*trans,j*-trans,k*trans);
        // La couleurs des cubes se remplie au fur et à mesure, en fonction de i,j,k
        // Ici, la constante permet de calculer l'intervalle, soit 5x51=255
        // On aura donc au départ (0,0,0) soit du noir
        // Puis (51,0,0) soit du rouge très désaturé
        // Jusqu'à (255,0,0) soit du rouge pur
        // Idem pour les deux valeurs suivantes
        fill(cstCouleur*i, cstCouleur*j, cstCouleur*k);
        // On dessine ensuite le carré
        box(cote);
        // On restaure la matrice de coordonnées précédente à chaque fois
        popMatrix();
      }
    }
  }
}
```

CUBE AU COULEURS DU WEB - Solution 2

J'ai mis en place une seconde solution qui fonctionne également et celle ci diffère au moment de remplir les cubes en couleur. Je n'ai plus utilisé d'intervalle mais j'ai défini un mode de couleur RVB avec seulement 6 nuances de couleurs. Et toujours pour optimiser j'ai définie un nombre de nuances de couleurs égale au nombre de cubes.

```
// Nombre de cubes à générer
int nbCubes;
// Gamme de couleur
int gammeCouleur;

// Il y a 6 cubes à générer
nbCubes = 6;
// La gamme de couleur est égale au nombre de cubes à générer
// On a ainsi l'ensemble des couleurs en fonction du nombre de cubes
// avec forcément plus ou moins de teintes selon le nombre de cubes
// Autrement dit, il n'y a plus 255 valeurs possibles, mais autant qu'il existe de cube, ici 6
gammeCouleur = nbCubes;

// Le mode de couleur est le RVB
// Gamme : les valeurs sont spécifiées entre 0 et 6 (nombre de cubes)
// Autrement dit, il y a autant de nuances qu'il y a de cubes
colorMode(RGB, gammeCouleur);
```

Ainsi, on peut directement remplir les cubes en couleurs comme suit :

```
// La couleurs des cubes se remplie au fur et à mesure, en fonction de i,j,k
// Ici, on est sur une Gamme 6
// On aura donc au départ (0,0,0) soit du noir
// Puis (1,0,0) soit du rouge très désaturé
// Jusqu'à (6,0,0) soit du rouge pur
// Idem pour les deux valeurs suivantes
fill(i, j, k);
```

ESPACE VIRTUEL - pushMatrix() et popMatrix()

Cet exercice n'est pas très éloigné du précédent, il consiste à créer des éléments en utilisant les transformations géométriques et les matrices. Dans cet exercice, nous allons, à partir d'un élément, en positionner d'autre à sa droite, à sa gauche, au dessus et au dessous.

On commence par préparé nos variables. Nous allons créer des cubes espacés entre eux.

```
// Côté des cubes
int cote;
// Espacement des cubes
int pas;
// Translation des cubes
int trans;
```

Nous allons maintenant définir les paramètres de la scène et initialiser les variables dans la méthode setup.

```
void setup() {
  // Taille de la scène
  size(400, 400, P3D);
  // Arrière plan de la scène
  background(128);
  // Couleur du contour des cubes
  stroke(190);
  // Longueur des côtés des cubes
  cote = 100;
  // Espacement entres les cubes
  pas = 15;
  // Translation des cubes
  // On translate donc d'une longueur égale au côté du cube, 100px, pour le positionner à côté, puis de la valeur pas pour le décaler de 15px, soit 100+15.
  trans = cote+pas;
}
```

Il reste à écrire la méthode *draw* qui va nous permettre de dessiner nos cubes.

Comme souvent, on ajoute un background pour nettoyer la scène. Ensuite, on va se positionner sur la scène, puis préparer la rotation de l'objet final en fonction de la souris. Ces étapes ont déjà été abordées dans le cube des couleurs pour le web.

```
void draw() {
  // Permet de nettoyer la scène
  background(128);
  // On se positionne dans la scène
  translate(width/2, height/2, -400);
  // On prépare la rotation de l'objet en fonction de la souris
  rotateX(mouseY*0.01);
  rotateY(mouseX*0.01);

  ...
```

On dessine ensuite un premier cube qui va servir de départ.

```
  // On prépare un remplissage en blanc
  fill(255, 0, 0);
  // On dessine un premier cube
  box(cote);
  // On enregistre la matrice de transformation courante
  pushMatrix();

  ...
```

On va maintenant pouvoir ajouter d'autres éléments en utilisant la translation, sans oublier de restaurer la matrice à chaque nouvelles transformations.

```
   // On se positionne sur l'axe x, en -trans, soit en -115. Donc à gauche du
rectangle
  translate(-trans, 0, 0);
  // On prépare un remplissage en vert
  fill(0, 255, 0);
  // On dessine un carré
  box(cote);
  // On restaure la matrice de transformation précédente
  popMatrix();
```

De cette façon, on peut créer n'importe quels éléments comme un ligne à l'aide de line ou une sphere à l'aide d'ellipse, et ce, n'importe où en utilisant la translation. On peut également créer des mouvement en faisant tourner certain éléments à l'aide d'une rotation. Mais il faut surtout ne pas oublier d'enregistrer la matrice puis de la restaurer à chaque fois.

PROGRAMME PERSONNEL, MINI CASSE BRIQUE

Dans ce programme personnel, j'ai repris des éléments vu tout au long des TPs. En créant un mini casse brique, j'ai du créer une barre qui suit le mouvement de la souris, déplacer une balle dans les limites de la scène et gérer son rebondissement sur la barre, créer des briques à l'aide de boucles, et gérer la collision entre la balle et chaque brique.

Dans un premier temps, j'ai créé les variables dont j'allais avoir besoin :

```
// Vitesse, Déplacement de la balle
int vBalleX;
int vBalleY;
// Coordonnées de la balle
int posBalleX;
int posBalleY;
// Coordonnées de la barre
int posBarreX;
int posBarreY;
// Déclaration du nombre de briques
int nb_briques=10;
// Déclaration des briques : tableau de briques
brique[] maBrique = new brique[nb_briques];
```

J'ai ensuite créer une class brique me permettant de définir les paramètres d'une brique à savoir, ses positions. J'ai également mis en place des méthodes relatives à la brique comme une permettant de déterminer sa forme et sa taille, et une autre de la faire disparaitre.

```
// Classe Brique
// Permet de définir les paramètres des briques
class brique {
  // Coordonnées, Positions des briques
  int posBriqueX;
  int posBriqueY;

   // Constructeur
   // Deux paramètres, les coordonnées des briques
   brique(int px, int py){
     posBriqueX=px;
     posBriqueY=py;
  }
```

```
// Fonction qui permet de dessiner les briques comme étant des rectangles
  void dessineBrique() {
    // Les positions sont des paramètres à définir
    // La taille des briques est 50x20
    rect(posBriqueX,posBriqueY,50,20);
  }

  // Fonction qui permet de faire disparaitre les briques
  void removeBrique() {
    // Les briques sortent de l'écran pour disparaitre
    posBriqueX=-100;
  }
}
```

Une fois mes variables définies et mes briques créées, je suis passé à la fonction setup qui permet de définir les paramètres généraux de la scène. J'ai notamment pu créer mes briques en initialisant ses positions.

```
// Fonction setup()
// Permet de définir les paramètres généraux
void setup() {
  // Taille de la scène
  size(600,400);
  // Arrière plan de la scène noir
  background(0);
  // Position de la balle initialement
  posBalleX = 100;
  posBalleY = 200;
  // Vitesse de la balle
  vBalleX = 2;
  vBalleY = 3;
  // Position de la barre initialement
  posBarreX = 300;
  posBarreY = 350;
  // Création des briques
  for (int i=0; i<nb_briques; i++){
    // 10 briques qui se place en x à intervalle de 50px puis en y=20
    maBrique[i] = new brique(50+(50*i),20);
  }
}
```

La méthode draw va permettre de nettoyer la scène et d'appeler toutes les méthodes relatives au jeu à savoir, dessiner et déplacer les éléments du jeu et faire rebondir la balle. Dans une boucle, j'ai pu dessiner les briques en appelant la methode dessineBrique. J'ai également tester la collision entre la balle et les briques au fur et a mesure.

```
// Fonction draw() qui s'éxécute tout le temps
void draw() {
  // Arrière plan à chaque fois pour nettoyer la scène
  background(0);
  // Fonction qui permet de dessiner les éléments du jeu
  dessiner();
  // Fonction qui permet de déplacer la balle et la barre
  deplacer();
  // Fonction qui permet de faire rebondir la balle
  rebondir();
  // Ajout des briques
  for(int i=0; i<nb_briques; i++){
    // Ajout des briques en appelant la fonction dessineBrique
    // Cette fonction dessine des rectangles de 50x20 au position renseignées lors de
la création des briques
    maBrique[i].dessineBrique();
    // Conditions si la balle entre en collision avec une des briques
    if(posBalleX>maBrique[i].posBriqueX && posBalleX-10<maBrique[i].posBriqueX+50 &&
posBalleY>maBrique[i].posBriqueY && posBalleY-10<maBrique[i].posBriqueY+20) {
      // La brique disparait
      maBrique[i].removeBrique();
      // Au moment de la collision, la balle rebondit et part en sens inverse
      vBalleY = -vBalleY;
    }
  }
}
```

Il ne reste plus qu'à mettre en place les méthodes du jeu.

- Dessiner
Permet de dessiner la balle et la barre, les briques ayant déjà été dessiner dans la boucle de la méthode draw.

```
// Fonction dessiner()
// Dessine les éléments du jeu
void dessiner() {
  // Barre du jeu blanche
  fill(255);
  // Sous forme de rectangle  de 100x10
  rect(mouseX,posBarreY,100,10);
  // Balle du jeu blanche
  fill(255);
  // Sous forme d'ellipse de 20x20
  ellipse(posBalleX,posBalleY,20,20);
}
```

- Deplacer
Permet de déplacer la balle en x et en y, ainsi que la barre en x en fonction de la souris.

```
// Fonction deplacer()
// Déplace les éléments du jeu
void deplacer()  {
  // Déplace la balle en x et en y
  posBalleX = posBalleX + vBalleX;
  posBalleY = posBalleY + vBalleY;
  // Déplace la barre sur x en fonction de la souris
  posBarreX = (mouseX);
}
```

- Rebondir
Permet de faire rebondir la balle dans les limites de la scène ainsi que sur la barre, si la balle atteint la limite basse, le jeu est perdu.

```
// Fonction rebondir()
// Permet de faire rebondir la balle dans les limites de la scène et sur la barre
// On inverse la vitesse de la balle à chaque fois
void rebondir() {
  // Limite droite, la balle va de gauche à droite
  if (posBalleX > width-10 && vBalleX > 0) {
    vBalleX = -vBalleX;
  }

  // Limite haute, la balle va de bas en haut
  if (posBalleY < 10 && vBalleY < 0) {
    vBalleY = -vBalleY;
  }

  // Limite gauche, la balle va de droite à gauche
  if (posBalleX < 10 && vBalleX < 0) {
    vBalleX = -vBalleX;
  }

  // Test si la balle entre en collision avec la barre
  if(posBalleX+10 > mouseX && posBalleX-10 < mouseX+100 && posBalleY+10 > posBarreY &&
posBalleY+10 < posBarreY+10)  {
    vBalleY = -vBalleY;
  }

  // Limite basse : si la balle descend trop bas, le jeu est perdu
  if (posBalleY > 390) {
    noLoop();
    println(«GAME OVER»);
  }
}
```

Le programme permet donc de faire rebondir une balle sur la barre, dans les limites de la scène et sur les briques tout en les cassant. Pour m'aider dans ce programme, j'ai suivi le tutoriel sur OpenClassRooms : http://fr.openclassrooms.com/informatique/cours/processing-1/td-pong

CONCLUSION

L'ensemble du cours et des applications sur les transformations géométriques m'a permis d'acquérir de nouvelles connaissances et de développer certaines compétences. Dans un premier temps, j'ai pu comprendre dans le détail le principe des transformations géométriques. Ce cours m'a notamment replongé dans des notions vues au lycée comme les vecteurs, ou le théorème de Pythagore et de Thalès dont j'ai pu voir un intérêt concret.

Même si tout n'est pas acquis, je pense avoir compris la majorité des notions et j'ai donc pu les appliquer tout au long de ce cours. J'ai ainsi pu renforcer mes compétences dans le développement, et même si l'AS3 se perd sur le web, je pense que le principe reste assez proche dans les autres langages.

D'ailleurs, j'ai également eu l'opportunité de découvrir un autre langage lors de ce cours, Processing. Et je rejoins ce que je disais précédemment, en ayant une bonne connaissance de l'AS3, il est assez rapide de s'adapter à un langage comme Processing.